**Chapter 6. Stream-Processing Patterns**

Stream-processing patterns evolved from event-driven architecture patterns. Event-driven architecture patterns revolve around event delivery and orchestration, whereas stream-processing patterns focus on how such events can be processed on the fly to extract meaningful information and take actions in real time. Without event-driven architecture patterns, we cannot implement stream-processing patterns in cloud native systems.

**What Is a Stream?**

A *stream* can be defined as a continuous sequence of events ordered by time. The stream consists of a name and version that uniquely identify it, such as *StockStream 1.0*. All events in a stream have a common message format and structure. For example, StockStream has a JSON format and contains *symbol*, *price*, and *volume* in its structure. Having a consistent format and structure allows events in the stream to be processed in an automated manner, using stream-processing systems. The stream version provides a way to safely modify the structure and evolve the stream over time.

**What Is Stream Processing?**

*Stream processing* is performing operations on events in motion. It can be as simple as a stateless service consuming events and transforming its event format, or as complex as storing and processing stateful data in memory with low latency and reliability.

In contrast to simple event processing, stream processing supports use cases in which events need to be handled in the order they are generated. Stream-processing patterns can also remember and use previous events when making a decision. For example, detecting if a stock price is continuously increasing over the last five minutes requires remembering previous events and processing them in order, both in real time.

**NOTE**

The term *real time*, in the context of stream processing, always refers to *near real time*. The system will try to provide results within milliseconds to a few seconds with best effort and always try to achieve low latency.

When building stream-processing applications, the stateless and stateful nature of the application can greatly influence the design. Therefore, we need to use a different set of patterns to preserve the state of the application. In this chapter, you will first learn how streaming data is processed to get meaningful output by using streaming data processing patterns, and then look at patterns for scaling, improving performance, and achieving reliability for both stateless and stateful stream-processing applications. Then we will discuss stream-processing technologies and how we can test, secure, monitor and observe, and continuously deploy stream-processing applications in the cloud.

**Streaming Data Processing Patterns**

*Streaming data processing patterns* focus on how we can generate useful output by processing real-time events through transformation, filtering, aggregation, and detecting meaningful sequences of events. These capabilities enable cloud native applications to process events on the fly with low latency.

A key performance consideration is avoiding heavy use of persistent data stores. In a cloud native application, the round-trip time of accessing the data store, and the potential for contention, can add significant processing latency to solutions. For some use cases, it is required, but as a general rule of thumb, it should be avoided.

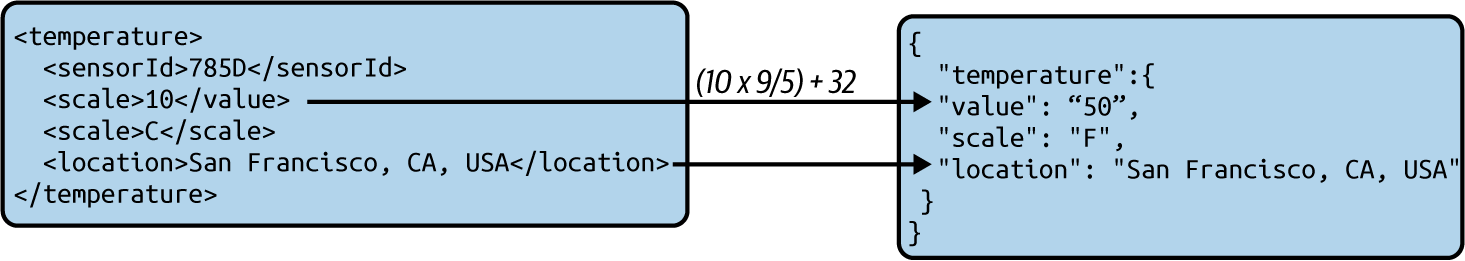
The following section dives into some of the key patterns related to stream processing in cloud native applications.

**Transformation Pattern**

The *Transformation pattern* helps transform events from an event source and publish them to another system with a different format, structure, or protocol.

**How it works**

This pattern maps the data of one event to another. For example, say we are to publish weather events to a third-party system that expects the events in JSON format with a particular structure ([Figure 6-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#xml_to_json_transformation)). The relevant data from the incoming event can be extracted and mapped to the new event format. We can achieve this by using JSON and XML libraries, or by using a graphical interface or SQL-based data-mapping approaches provided by stream-processing technologies.



**Figure 6-1. XML-to-JSON transformation**

These transformations are often achieved purely with the information contained in the incoming event. But at times these transformations need other patterns, such as the Windowed Aggregation pattern, which we’ll discuss shortly.

**How it’s used in practice**

This pattern can be applied to cloud native applications in any programming language. This pattern also can be applied by systems like service buses and stream processors.

**Message transformation**

Messages can be transformed by using various techniques, such as via code with traditional programming languages and through specialized applications that perform data mapping. These applications include service buses and stream-processing systems that can run on the cloud, such as Apache Camel, KSQL, Amazon Kinesis, and Azure Stream Analytics.

Let’s walk through an example of message transformation. Imagine that when taxis complete their journey, they publish JSON events containing relevant trip information. We need to extract pickup and drop-off locations and construct XML events for analytical applications so that they can analyze user movements and predict demand.

[Figure 6-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#converting_a_json_message_to_xml_by_usi) illustrates how we can use a standard cloud native application to consume the incoming JSON event and convert it to an XML event.

We can use JSON path expressions along with JSON libraries to extract data from the JSON event and then use that data to construct the XML messages. One of the easier ways of constructing messages is by using event templates and filling in the relevant data. Simple text templating libraries such as Mustache can be used to populate the templates and generate events. Most important, these events need to be validated for correctness by using XML libraries before sending them out. For instance, if the content of a field contains an opening XML tag such as <item>, blindly inserting them into the template will alter the final event structure and generate malformed events.



**Figure 6-2. Converting a JSON message to XML by using JSON path and text templating**

**Protocol switching**

When working with partners and third-party teams, sometimes different teams will use different, noncompatible message brokers. One team might use Kafka for its message processing, while another uses Apache ActiveMQ, for instance. We cannot simply send events from one to another without some kind of conversion. Here, we use an intermediate application that consumes events from AMQP and deserializes them. Then it serializes those events as Kafka events and publishes them to Kafka.

Protocol switching alone does not require data mapping, so it can be implemented via a simple cloud native application by using the appropriate protocol libraries for both event consumption and publishing.

**Considerations**

This pattern is especially useful when we are working with applications that are managed by partner teams, and we need to perform transformations to allow our cloud native applications to interoperate.

For stateless transformations, the cloud native applications can be scaled horizontally without any issues. We can use serverless compute options such as Amazon Lambda or Azure Functions for these use cases.

When these transformations are stateful—for example, when we need the Windowed Aggregation pattern to calculate the average temperature over the last hour—these systems cannot be simply scaled horizontally. The Sequential Convoy pattern will show us how to partition and scale these applications.

**Related patterns**

The Transformation pattern can be combined with other stream data processing patterns, as data transformations can be required for incorporating results of those patterns, such as enriching events with aggregated data.

**Filters and Thresholds Pattern**

Sometimes we need to filter events based on given conditions, or allow only events with values that fit within a given threshold range. The *Filters and Thresholds pattern* is useful for extracting only the relevant events we need.

**How it works**

Users provide conditions that match against the incoming events. These conditions can include exact string matches, substring matches, regular expressions, or threshold ranges when it comes to numeric values with comparison operations such as <, <=, >, >=, and ==. Often more than a single condition is required, so those conditions are consolidated by using the AND, OR, and NOT logical operations and parentheses to generate more-complex filter conditions.

If we are processing a real-time stream of car sales and are interested in only 2010 or newer Toyota vehicles, we can define a filtering condition as shown in [Figure 6-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#filtering_car_events_based_on_brand_and) to emit only events that satisfy the condition.



**Figure 6-3. Filtering car events based on brand and year**

This pattern extracts and processes the relevant data from the input event stream by using data-mapping techniques discussed in the [“Transformation Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#transformation_pattern).

**How it’s used in practice**

This pattern can be applied to cloud native applications written in any programming language, and by systems like service buses and stream processors.

**Filter events by category**

Often we are interested in only certain types of events for processing. Take, for example, handling asynchronously published local and international shipment events distinctly in an ecommerce platform. In this case, when possible, use subscription filters provided by message brokers to filter only the relevant type of data for processing. But when that is not possible, we recommend implementing an intermediate microservice or serverless function to filter and publish only the relevant events. This also improves security and eliminates potential misuse of data, especially when the data is published to third parties.

**Scenario: Apply a threshold for alerting**

Sometimes we’re not interested in certain events, and processing everything at all times is not computationally feasible. In this case, it is essential to filter only the most critical data based on a threshold. For example, in a banking use case with hundreds of transactions performed every minute, performing human verification on all events to detect fraud is not possible.

Banks filter only high-value transactions for human verification. The filtering condition may contain not only the value of the transaction, but also the location of the transaction and whether it took place online or in a store. [Example 6-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#query_six_onedot_request_for_human_inte) shows a sample query used to determine if a transaction is high risk. Such transactions are then sent for human verification so that they cross-check with the card holder if the given transaction is legitimate.

**Example 6-1. Request for human interaction only for online transactions of more than $1,000 in the United States and for all non-US transactions greater than $500**

(amount > 1000 AND place == "USA" AND isOnline == true)

OR (amount > 500 AND place != "USA")

**Considerations**

This pattern not only allows cloud native applications to extract relevant events for processing but also reduces their load by dropping events that are irrelevant or lower priority.

It is important to note that modern message brokers such as Kafka now natively support this functionality, allowing cloud native applications to subscribe to their topics with a filter condition. This also avoids running additional containers just for filtering. This option is not always available, especially when publishing events to third-party systems.

Filters can be implemented as stateless microservices and deployed in front of any other cloud native application to filter and pass only the events that are relevant. We can also readily leverage serverless compute options such as Amazon Lambda and Azure Functions to implement this pattern.

**Related patterns**

The Filtering and Thresholds pattern can be applied with all the other stream data processing patterns, as we often need to filter events for those patterns (for example, to aggregate only a particular type of event).

**Windowed Aggregation Pattern**

The *Windowed Aggregation pattern* enables us to analyze a collection of events based on a condition. Here, aggregation analysis can include operations like summation, minimum, maximum, average, standard deviation, and count, and the window defines the collection of events used for aggregation.

These windows can be based on the time or event count, such as the last five minutes or the last 100 events. These windows may also have behaviors such as sliding or batching, defining when events are added and removed from the window.

This pattern enables us to aggregate data on the fly and make time-critical business decisions within milliseconds.

**How it works**

Understanding how windows operate is fundamental to understanding the behavior of this pattern. Let’s look at some of the most common windows—such as length sliding, length batch, time sliding, and time batch—that are supported by most stream-processing systems. The aggregation operations are performed on top of these windows, as windows limit the number of events that need to be considered for aggregation, and the aggregation output is emitted as a stream for further processing.

Let’s explore how these windows operate. For example, a time sliding window of one minute, considers only the events that occurred during that last minute. Events are added and removed from the window as time progresses. This window emits aggregations of all the events within it upon every addition or removal of an event from the window. For implementation optimizations, instead of adding events as soon as they arrive, some stream-processing systems require us to provide a sliding interval, defining how often the window will slide—in other words, how often the events will be added and removed from the window. For example, a one-minute sliding window that has a sliding interval of one second will slide by one second at a time. During that slide, the window will add all the new events that arrived in the preceding second and remove all the events that were from the oldest second. In this case, the aggregations are emitted every second as events are added and removed.

As an example, let’s say we’re continuously monitoring a purchase stream, to count the total number of units ordered over the preceding minute, using a one-minute sliding window with a one-second sliding interval, as shown in [Table 6-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#summation_of_units_ordered_over_a_one_m).

| **Time in milliseconds** | **Input: number of units ordered in each purchase event** | **Output: number of units ordered over the preceding one minute** |
| --- | --- | --- |
| 5:30:20 000 (start of first 1 minute) | - | - |
| 5:30:20 007 | 5 | - |
| 5:30:20 115 | 6 | - |
| 5:30:20 545 | 11 | - |
| 5:30:21 000 (start of second 1 minute) | - | 0 + (5 + 6 + 11) = 22 |
| 5:30:21 100 | 2 | - |
| 5:30:21 393 | 14 | - |
| 5:30:22 000 (start of third 1 minute) | 4 | 22 + (2 + 14 + 4) = 42 |
| 5:30:47 560 | 7 | - |
| 5:30:48 000 | - | 42 + 7 = 49 |
| 5:30:23 000 (start of fourth 1 minute) | - | - |
| 5:30:24 000 (start of fifth 1 minute) | - | - |
| ... | ... | ... |
| 5:31:19 000 (start of 60th 1 minute) | - | - |
| 5:31:20 000 (end of first and start of 61st 1 minute) | - | - |
| 5:31:20 345 | 8 | - |
| 5:31:21 000 (end of second and start of 62nd 1 minute) | - | 49 + 8 – (5 + 6 + 11) = 35 |
| 5:31:21 500 | 15 | - |
| 5:31:22 000 (end of third and start of 63rd 1 minute) | 13 | 35 + (28) – (20) = 43 |
| Table 6-1. Summation of units ordered over a one-minute time window that slides each second | | |

In contrast, a length batch window of size 4 will process and emit aggregations after every fourth event arrives. Continuing with the previous example of monitoring a purchase stream, as shown in [Table 6-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#summation_of_units_ordered_over_a_lengt), the new purchase events are added to the window as they arrive, but the aggregated unit count is emitted only when the fourth event is added to the window. At that time, the window all events stored in the window will also expire.

| **Time in milliseconds** | **Input: number of units ordered in each purchase event** | **Output: number of units ordered during the last four purchases** |
| --- | --- | --- |
| 5:30:20 007 | 2 | - |
| 5:30:20 115 | 6 | - |
| 5:30:20 545 | 4 | - |
| 5:30:21 000 (end of first batch) | 3 | (2 + 6 + 4 + 3) = 15 |
| 5:30:21 100 | 2 | - |
| 5:30:21 393 | 14 | - |
| 5:30:22 000 | 7 | - |
| 5:30:47 560 (end of second batch) | 5 | (2 + 14 + 7 + 5) = 28 |
| 5:30:48 000 | 4 | - |
| 5:31:20 345 | 7 | - |
| 5:37:26 353 | 3 | - |
| 5:38:21 500 (end of third batch) | 1 | 15 |
| ... | ... | ... |
| Table 6-2. Summation of units ordered over a length batch window of size 4 | | |

The aggregation operations can be applied to any of these windows, and the point where the aggregation emits results depends on the type of window chosen. For example, a length batch window of size 10 produces aggregated results for every 10 events, and a time sliding window of five minutes with a one-second sliding interval will emit output every second.

The aggregation functionality can also be combined with group by, having, order by, and limit operations (similarly to SQL), to group the aggregations by a field, filter, sort, and limit the output as per our needs.

Finally, it is important for this pattern to use the Transformation pattern to map the aggregated results into the output.

**How it’s used in practice**

The Windowed Aggregation pattern is stateful, meaning it stores data related to the events in memory. Therefore, when designing noncritical use cases such as monitoring that tolerates data loss, we can implement this pattern on any cloud native application. But when the use case requires reliable event processing, we need to combine this pattern with the reliability patterns, which we discuss later in the chapter.

**Aggregate events over time**

Some use cases require us to aggregate multiple events over a period of time. For example, let’s consider a fraud detection use case: instead of analyzing individual transactions, we want to learn the top 10 users by finding the total amount that they have transacted during the last 10 minutes. This detects whether someone is splitting a large sum of money and transferring it by using lots of small transactions.

We can use a 10-minute time sliding window with a 1-second slide interval, a sum aggregator to sum the transactions, a group by aggregator to group the aggregations by the user, and finally the sort and limit to extract only the top 10 users out of that. [Example 6-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#query_six_twodot_aggregating_events_by) shows this.

**Example 6-2. Aggregating events by transaction value over the last 10 minutes and outputting the top 10 users with their total amount transacted**

select userName, sum(transactionValue) as totalTransaction

from InputStream

window time (10 min, 1 sec)

group by userName

order by totalTransaction desc

limit 10

In this example, system downtime may cause business impact. Therefore, we have to apply reliability patterns such as the Two-Node Failover pattern, covered later in this chapter, to make sure that accurate aggregation calculations are continuously emitted for decision making.

**Aggregate events over length**

Sometimes the number of events is an important aspect of the aggregation, and those cannot be modeled with time. Say we want to receive an alert when the server rejects three consecutive requests. We can use a length sliding window to identify whether it has rejected the last three events it has received. See [Example 6-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#query_six_threedot_determining_whether).

**Example 6-3. Determining whether the last three consecutive requests were rejected**

select serverId, sum(isRequestRejected) as totalRejectedRequests

from InputStream

window length (3)

having totalRejectedRequests == 3

In this case, isRequestRejected will contain 1 when the server rejects a request and 0 when it has served the request successfully.

We assume that all the events processed by the query are produced by a single server. Otherwise, we need to combine this pattern with the Sequential Convoy pattern (discussed later in this chapter) to partition the query to process events arriving from multiple servers.

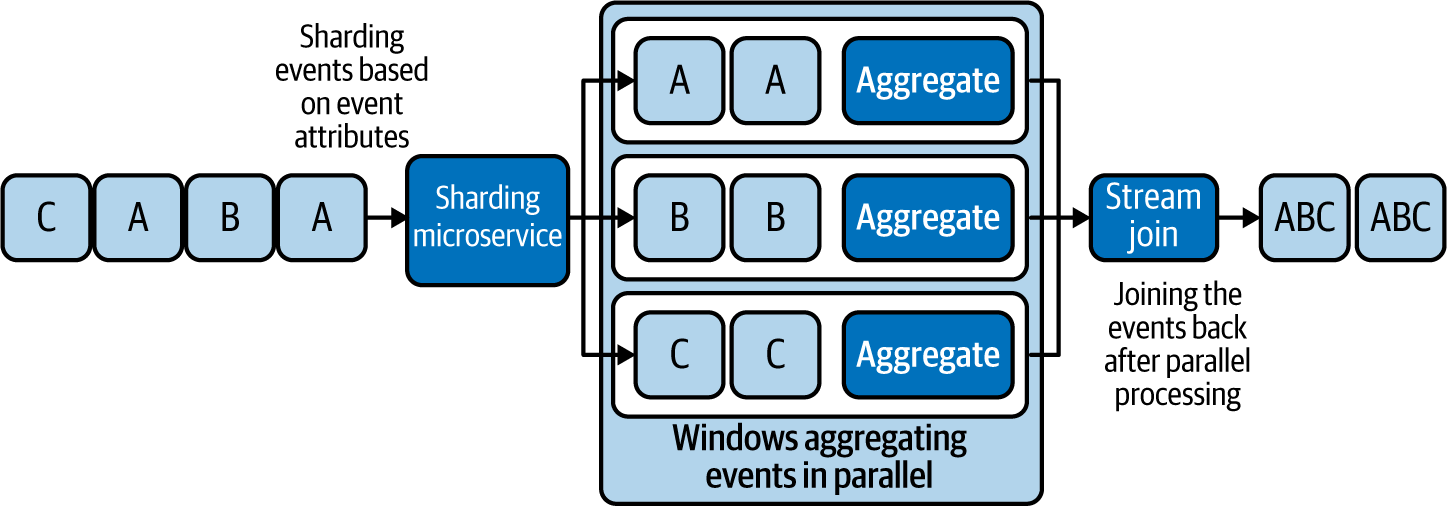
If the service is not critical for the business, system downtime may not cause business impact. Therefore, we don’t need to worry about preserving the window state, and so there is no need to use reliability patterns.

**Considerations**

The most important aspect of this pattern is that it is stateful. Windows rely on multiple events, and a system failure or restart can cause those events to get lost, causing the aggregations to emit inconsistent results. When aggregations are not used for mission-critical use cases, it may be acceptable to lose those events during system failures or restarts. In this case, some aggregation outputs will not be published or will be inaccurate. But when the aggregation outputs are critical, we can apply reliability patterns (discussed later in this chapter) to make sure that we are appropriately rebuilding or recovering the state after a failure or restart.

It is also important to consider that we cannot implement all types of aggregations with high accuracy and efficiency. For example, we can use windows to model the mean (average), but not the median. The mean needs only the sum and the count of events in the window, and techniques can be used to progressively alter these values as events are added and removed from the window. This enables us to rapidly compute the average (sum/count) by not iterating through all the events in that window. But on the other hand, to calculate the median, we need to iterate through all the events in the window. This will not only add latency to the calculation, but persisting all events requires more space, which becomes more problematic as windows get larger.

This now brings us to scaling of these operators. It is vital that we design the system to withstand high load and scale on demand. Because windows are collections of events, the most effective way of scaling them is by sharding. [Figure 6-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#sharding_events_based_on_event_attribut) illustrates splitting the incoming events into different windows, aggregating the events in isolation, and then using the Stream Join pattern (which we detail next) to build bigger aggregations. We discuss scaling based on sharding in the [“Sequential Convoy Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#sequential_convoy_pattern).



**Figure 6-4. Sharding events based on event attributes, processing them in parallel, and joining the results**

**NOTE**

It is also possible to perform aggregations without a window. Consider that the window spans from the service startup to the current time. Aggregations would reflect all the events that occurred during that period. But it is important that we implement these window aggregations to operate with constant space complexity (consumption of memory not depending on the number of events processed); otherwise, the system could run out of memory.

Implementing the Windowed Aggregation pattern from scratch can be time-consuming and error prone. We recommend that you use cloud native stream processors or stream-processing libraries such as Esper or Siddhi when possible to fulfill these use cases.

**Related patterns**

The following are related to the Windowed Aggregation pattern (all are covered in this chapter unless otherwise noted):

*Transformation pattern*

Appropriately maps the aggregation to the output.

*Reliability patterns*

Help make the window and aggregation state survive system failures.

*Sequential Convoy pattern*

Allows aggregations to be performed in parallel based on shard keys. This not only helps scale aggregation processing, but also allows us to aggregate different types of events in isolation and produce aggregations per event type.

*Service Orchestration pattern*

Splits the events by different shard keys for processing. This pattern is described in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern).

*Stream Join pattern*

Aggregates results from different shards.

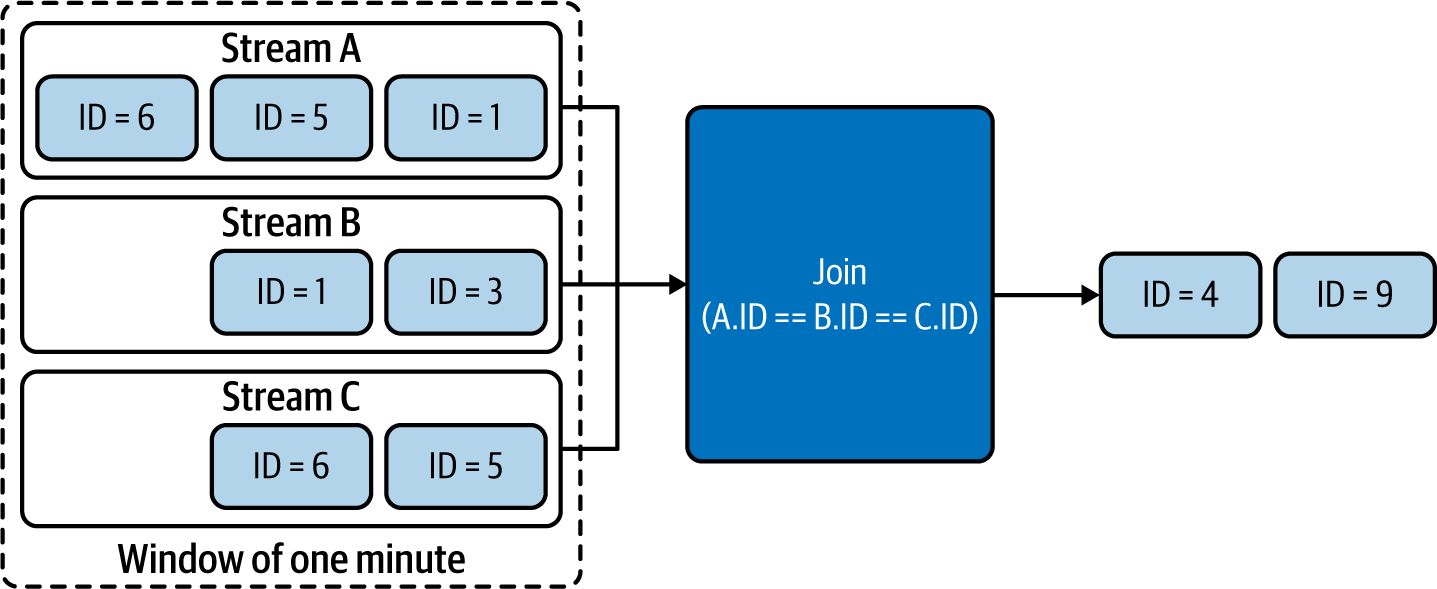
**Stream Join Pattern**

The *Stream Join pattern* resembles the join of SQL tables and enables us to join events from multiple streams with different schemas.

**How it works**

This pattern works by defining a condition to identify the joining events. This condition will pick attributes from each joining event stream and define the condition under which they should be joined. This can be a simple equal condition, like joining events from all event streams having the same ID, or it can be more complex.

The join should also define a buffer that determines how long events should wait for corresponding events to arrive from other event streams. This buffer period can be common across all streams or can vary among streams. Most stream-processing systems define this buffer period via windows that you learned about in the previous section. In [Figure 6-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_join_based_on_events_that_have_a), we see the buffer defined as a one-minute window.



**Figure 6-5. Stream Join based on events that have arrived during the last minute**

In this example, we assume that events with IDs 4 and 9 arrived in all streams (A, B, and C) during the last minute; those events are joined and emitted. Events with IDs 6, 5, 1, and 3 are still waiting in the window for their corresponding events with the same event ID to arrive in all streams so that the join condition can be satisfied.

Finally, as in the Windowed Aggregation pattern, it is important for this pattern to use the Transformation pattern to map the joining events and their attributes to the output.

**How it’s used in practice**

The Stream Join pattern is stateful, as it buffers events for the join. Like the Windowed Aggregation pattern, this one can be implemented in any cloud native application as long as the use case is not business critical and can tolerate event loss. But when event loss is not acceptable, this pattern should be applied along with reliability patterns, so the application can withstand system failures and restarts without event loss.

**Scatter and gather**

In *scatter and gather*, we process the same event in parallel, performing different operations, and finally combine the results so all event outputs can be emitted as a single event. This is one of the most common scenarios for using this pattern.

For example, let’s consider a loan application process. The loan application can be initiated by an event that contains a unique loan application ID. Operations for this event—credit check, address verification, and identity verification—can be processed in parallel. But at the end, the outputs of all three operations need to be joined in order for the bank to make a decision on whether the applicant should be granted a loan.

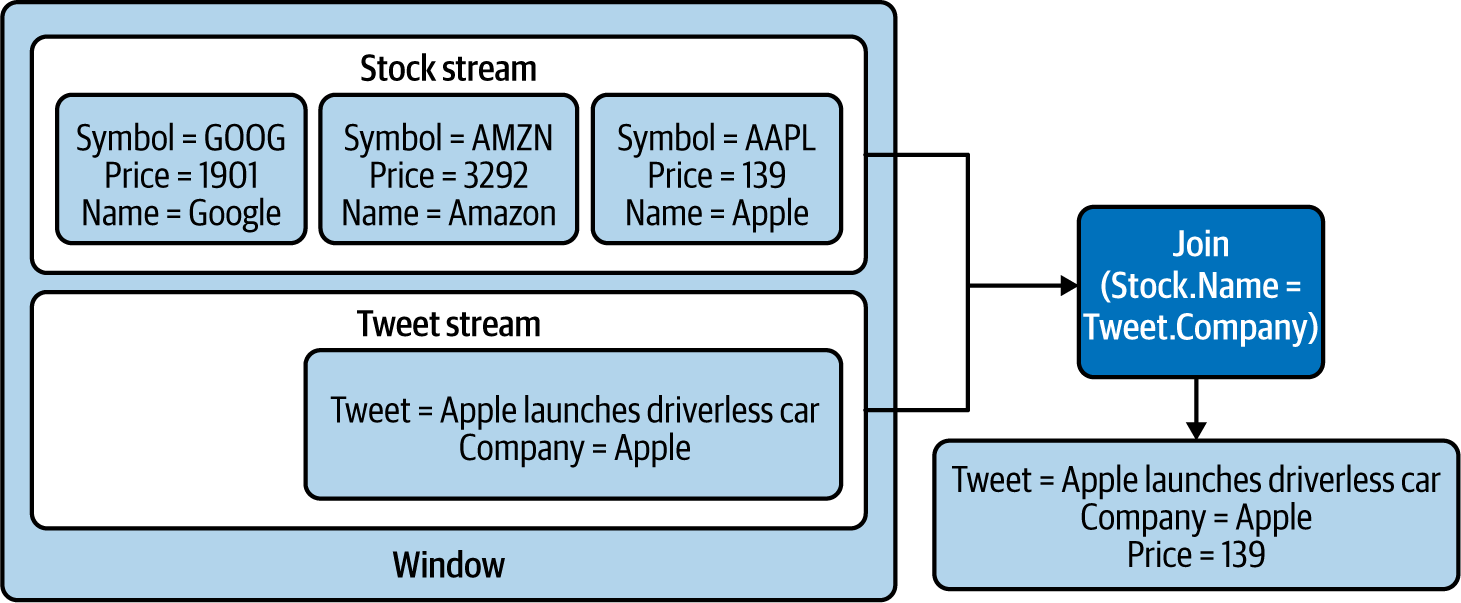
The results of all the parallel operations contain the same loan application ID, and we can implement a microservice to perform join operations based on that ID. To generate an output, the join operation will wait for all three corresponding events to arrive. It is critical that all the parallel processors send a response event, whether the response is a success or failure, as this will help perform the join in a more deterministic way. When processors do not send events because of errors or network failures, we should have a defined strategy for handling that scenario. The buffer period we define can help identify missing events and still emit the joint event with missing results, so that when possible, a decision can be made from the partial results (for example, if the partial results have a failure, the loan is rejected, or if we cannot determine a decision, a reprocessing of the data is initiated).

**Join various types of events**

The Stream Join pattern can also be used to join various types of events based on a defined condition and a window, as discussed in the [“Windowed Aggregation Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#windowed_aggregation_pattern). Let’s take an example of identifying stock prices at times when certain tweets are published.

As depicted in [Figure 6-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#joining_stock_prices_with_tweets_based), we use a window to store all the latest stock prices from the Stock stream, and when we identify a new tweet containing, say, a company name such as Apple, we join that event against the stock events in the window. According to the join condition, the tweet will match the stock event having the same company name (in this case, the event with symbol AAPL), and the joined event is emitted as an output with the latest stock price.

This kind of join can be an inner, a full-outer, a left-outer, or a right-outer join, as in SQL queries. During outer joins, we can replace event attributes with null when we cannot identify matching events in the window.



**Figure 6-6. Joining stock prices with tweets based on company name**

**Considerations**

Join is a stateful operation; it needs to wait for all matching events to arrive before it makes a valid join. When event loss cannot be tolerated, we use reliability patterns to ensure that events are preserved across system failures and restarts.

But for simple scenarios such as scatter and gather, we can directly read events from message brokers and defer acknowledgment until those events are successfully joined. With this approach, we do not lose those events upon a system failure or restart, as the message broker will republish those. More details about this approach are discussed in [“Replay Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#replay_pattern).

Joining many events during a long time period can be challenging, as systems may suffer from increased space requirements and increased processing times. In this case, we recommend the Sequential Convoy pattern discussed later in this chapter to shard events based on the joining attributes. This will parallelize joining and ensure that related events fall into the same shard so they can be joined successfully.

**Related patterns**

These patterns (covered in this chapter) are related to the Stream Join pattern:

*Transformation pattern*

Appropriately maps joining event attributes to build the output.

*Reliability patterns*

Helps the joint state survive system failures.

*Sequential Convoy pattern*

Scales joins by performing them in parallel by allowing relevant joining events to fall into the same shard.

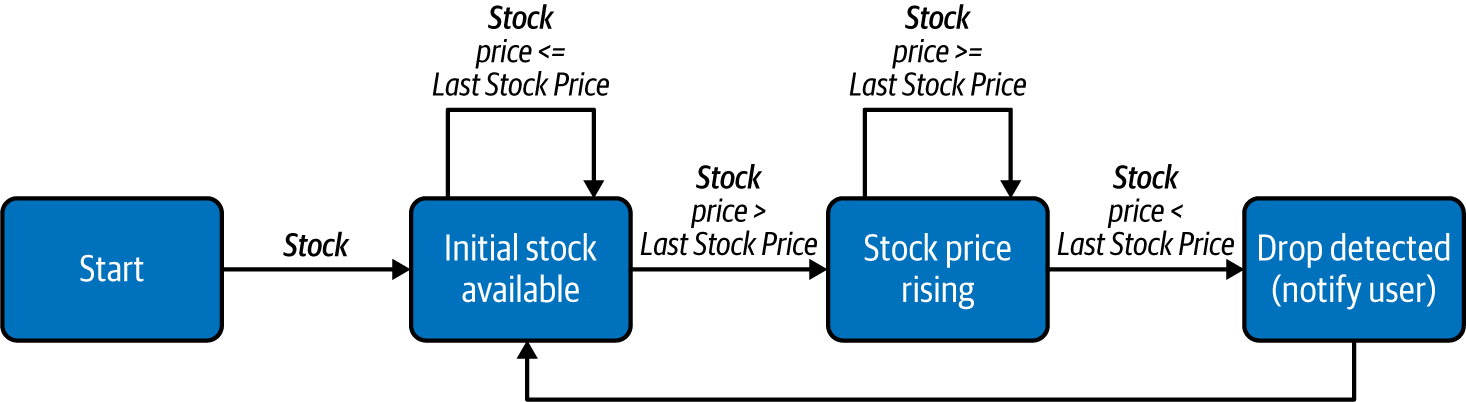
**Temporal Event Ordering Pattern**

The *Temporal Event Ordering pattern* is unique for stream processing. It tries to detect various interesting complex event occurrences by identifying patterns based on event arrival order. The pattern can also detect occurrence and nonoccurrence of incidents based on events emitted by various systems.

**How it works**

This pattern works on the concept of nondeterministic finite-state machines: the application state changes based on the input event and the current application state. The possible state transitions can be represented as a state graph that traverses from one state to another until it reaches either a success or fail state. Upon reaching the success state, the user is notified, as it means the expected events have occurred in order.

[Figure 6-7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#using_the_temporal_event_ordering_patte) shows an example of the Temporal Event Ordering pattern. We detect a continuous stock price increase followed by a single drop, and the user will be notified as soon as the first drop is detected.



**Figure 6-7. Using the Temporal Event Ordering pattern to detect a continuous stock price increase followed by a single drop**

This pattern can also be used to identify sequences of events that are immediately followed by one another or scattered randomly among other events. We can also use this to detect the nonoccurrence of events by combining state transitions with time-outs.

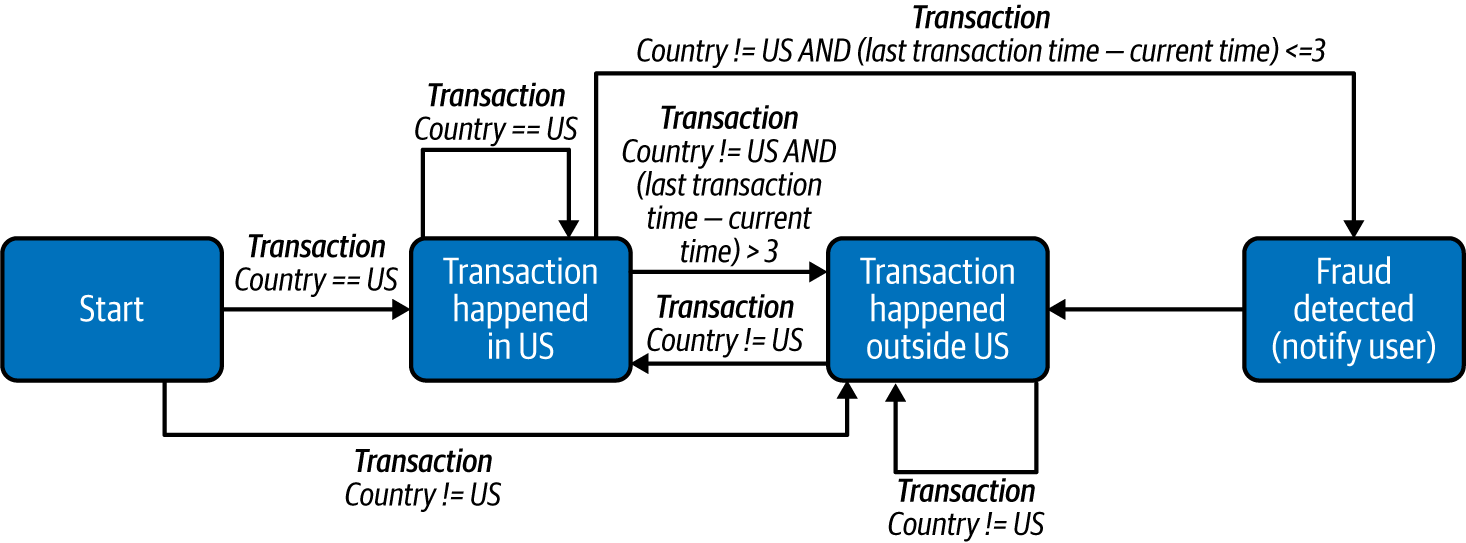
Use cases such as stock monitoring most often require the event sequence to be detected repeatedly. To achieve this, a new state machine instance should be initiated upon each event arrival that triggers the initial state of the state machine. In the preceding example, the event that triggered the final state can be used as the initial event for a new instance of the state machine.

**How it’s used in practice**

Like the Windowed Aggregation and Stream Join patterns, this pattern should also be combined with reliability patterns to preserve data loss during system failures and restarts. Furthermore, as event arrival order is critical for the success of this pattern, we recommend using patterns like Buffered Event Ordering (discussed later in this chapter) to guarantee ordering of events before processing them.

**Detect sequence of event occurrence**

The most common use of this pattern is for identifying an incident by having a sequence of events happen in a prescribed order. Let’s consider an example of detecting fraudulent credit card transactions. A fraudster can copy and use credit cards without the card holder being aware. But this kind of fraud can be detected by using predefined rules, such as a transaction happening in the US followed by another transaction happening on the same credit card outside the US within three hours. [Figure 6-8](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#state_machine_for_detecting_transaction) depicts using a state machine for the detection.



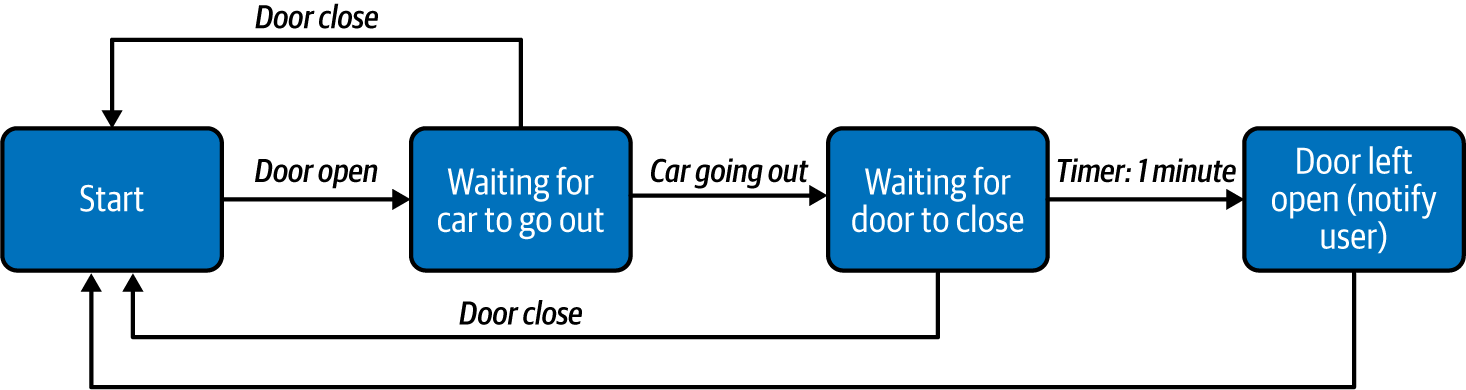
**Figure 6-8. State machine for detecting transaction fraud**

In this example, a new state machine instance should be created for each credit card, which can then keep track of the previous transaction to determine whether the next transaction is occurring outside the US within three hours.

**Detect nonoccurrence of event**

Now, let’s say we want to identify an incident by an expected event *not* occurring. These are commonly used for detecting erroneous situations, such as notifying a homeowner that their garage door has been left open.

The user needs to receive a notification if the garage door is left open for one minute after the car drives out. This pattern expects the door-close action to take place within one minute of the car leaving and notifies the user if the door does not close within that time frame (the nonoccurrence of the event). [Figure 6-9](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#detecting_the_garage_door_left_open_for) depicts how this detection can be designed.



**Figure 6-9. Detecting the garage door left open for more than 1 minute after the car leaves the garage**

This requires a timer task to be initiated to keep track of the time, and that timer should be cancelled as soon as the door is closed, to prevent an erroneous notification from being sent.

**Considerations**

As state machines are inherently stateful, this requires applications to rely on reliability patterns (discussed later in this chapter) to persist their states across system failure and restarts. Also, we should ensure that each cloud native application has enough in-memory space to maintain the state machines. In addition, we should apply the Sequential Convoy pattern to distribute events to various nodes so that the sequence  matching can be scaled and parallelized, while making sure all relevant events for a successful match are still routed to the same node.

One of the other important aspects of this pattern is that it requires events to be processed in the order they are generated. Though it is not possible to always determine relative ordering of events, correlating and ordering events based on event-generation time can still help overcome out-of-order events that happened during transmission. We recommend you use the Buffered Event Ordering pattern to guarantee ordering of events if they can become out of order during transmission.

As with earlier patterns discussed in this chapter, implementing this pattern can be time-consuming and error prone. Therefore, we recommend you use cloud native stream processors to fulfill these use cases. Stream processing systems like Azure Streaming Analytics, Apache Spark, Apache Flink, Esper, and Siddhi are some that can provide this functionality by default. We recommend building this pattern from scratch only when such systems are not available in your environment.

**Related patterns**

The following are related to the Temporal Event Ordering pattern; all are covered in this chapter:

*Transformation pattern*

Appropriately maps the matched events in the sequence to generate a meaningful output.

*Reliability patterns*

Helps state machines survive system failures.

*Sequential Convoy pattern*

Scales sequence matching by performing it in parallel by allowing relevant events to fall into the same shard.

*Buffered Event Ordering pattern*

Orders events based on event-generation time to facilitate correct behavior of this pattern.

**Machine Learner Pattern**

We can use machine learning models in real time to generate predictions and automate decision making. Machine learning models can be prebuilt to produce predictions without updating themselves based on new input events. Online machine learning models can produce predictions while continuously learning, based on new incoming events, whether or not they’re pre-generated, making our cloud native application much more intelligent.

**How it works**

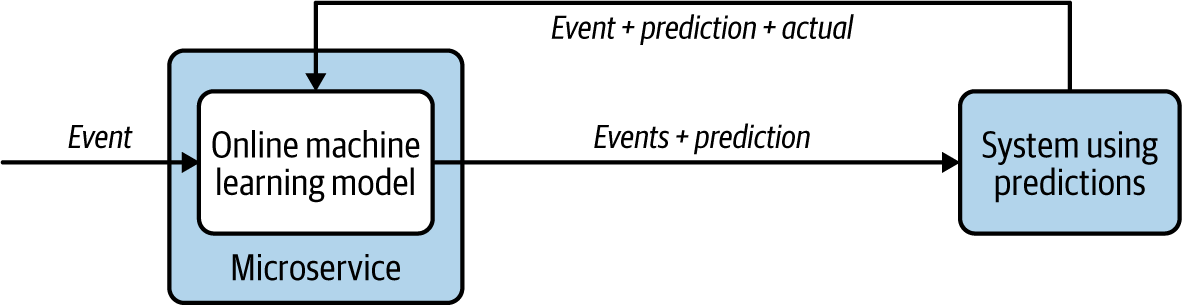
We can generate predictions in cloud native applications in two ways: by executing prebuilt machine learning models and by using online machine learning models. We discuss these approaches in detail next.

**Prebuilt machine learning models**

These models can be generated by a data scientist using data processing tools and machine learning frameworks such as Apache Spark, TensorFlow, or even Python. Some of these models can be imported into running applications via technologies such as Predictive Model Markup Language (PMML), and we can query them on the fly to generate predictions. We can also run them as separate cloud native applications and call them via APIs. Because these models are prebuilt and cannot adapt based on new incoming events, we need to update them periodically to maintain and improve their prediction accuracy.

**Online machine learning models**

These are models that tune themselves based on the information they receive as they produce predictions. In some cases, the models require a feedback loop with the results from their previous predictions so that they dynamically train themselves. These models can be embedded into applications or run as separate microservices. [Figure 6-10](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#microservice_with_an_online_machine_lea) shows a microservice with an online machine learning model that also continuously updates itself based on past results.



**Figure 6-10. Microservice with an online machine learning model using a feedback loop to continuously learn**

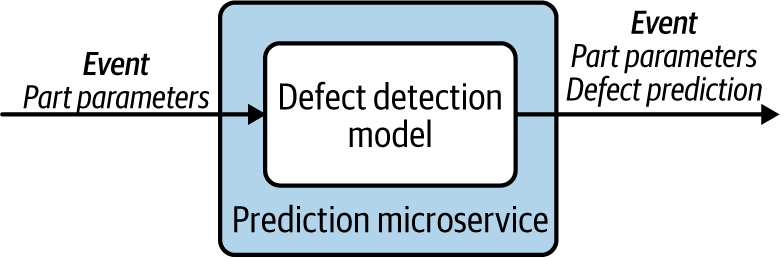
**How it’s used in practice**

Machine learning has now become an integral part of many applications, and cloud native applications should also be well equipped to incorporate them. One common way of integrating machine learning models is to deploy them as individual microservices and make service calls. This is no different from calling other services, as covered in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern). Alternatively, machine learning models can be embedded into the applications, which can continuously produce predictions based on incoming events. Some scenarios using this pattern are described next.

**Predict based on prebuilt machine learning models**

Using a prebuilt machine learning model is ideal when we have abundant historical data, and when the prediction pattern does not change with new events. Let’s consider an example of automating detection of defective parts in a production line. Detecting defects early in a production line can reduce cost.

A microservice can be deployed with a prebuilt linear regression model to examine the parameters of parts and detect any defects, and to allow the parts that do not have defects to progress along the pipeline ([Figure 6-11](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#running_a_prebuilt_machine_learning_mod)). If the manufacturing happens in a controlled environment with the same input materials, temperature, and machinery, the prediction will be accurate for a longer period, and we don’t need to update the prebuilt models as often.



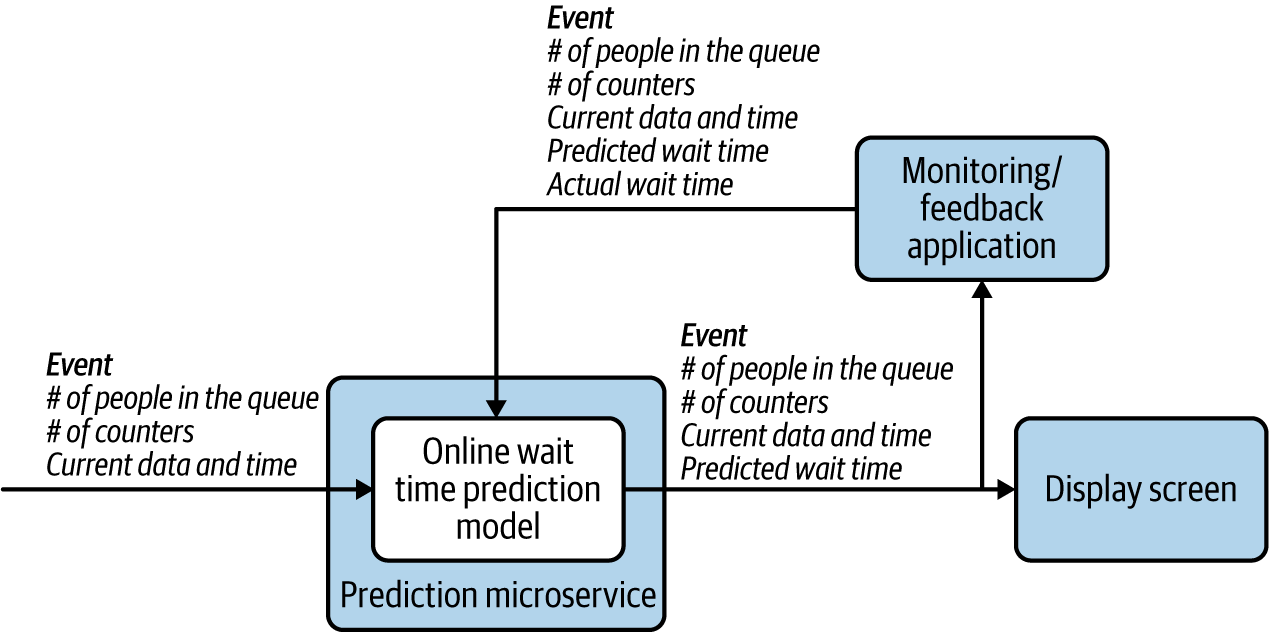
**Figure 6-11. Running a prebuilt machine learning model to detect defects**

**Continuous learning with data**

We recommend using continuously learning online machine learning models if we expect them to learn new behaviors (as we receive new data) in the future. To give you an example of this type of scenario, let’s say we want to predict the expected wait time for an airport security scan, and display that wait time on screens throughout the airport.

We can use a machine learning model to make predictions based on the number of people waiting in line, available security check counters, and the time taken for them to finish the check. But because of ever-changing environmental effects, such as insider information about potential security threats, using a prebuilt machine learning model with historical data will not produce accurate predictions.

But using an online machine learning model is beneficial, as it learns from its feedback and adjusts its predictions in real time ([Figure 6-12](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#emitting_accurate_wait_time_predictions)). For this model to work successfully, we need an application that provides input, continuously feeding in the number of passengers waiting in the line, the available counters, as well as the actual time taken. This will enable the model to continuously emit more-accurate predictions on the screens.



**Figure 6-12. Emitting accurate wait-time predictions through feedback**

**Considerations**

Prebuilt machine learning models are much simpler to implement and generally have high accuracy when we have enough data. At the same time, they cannot adapt to new trends. For these cases, online machine learning algorithms perform much better. But these algorithms can fluctuate in accuracy, because most are constrained by the amount of data they can store in memory and therefore have limitations on how much data they can learn. This compromises their ability to produce data with high accuracy. Therefore, we recommend combining prebuilt and online machine learning algorithms where possible, so you can override the predictions of the prebuilt models when you have higher accuracy from the online machine learning models.

When using prebuilt machine learning models, it is important to update them periodically. Over time, the model accuracy can degrade because of changes in circumstances. In cloud native applications, these models can be embedded, so updating the application version along with a newer model provides an easy deployment path.

Online machine learning models store what they’ve learned in memory. We recommend using reliability patterns on those cloud native applications so they recover state across system failures and restarts.

**Related patterns**

The following patterns, covered in this chapter, are related to the Machine Learner pattern:

*Transformation pattern*

Appropriately maps the predictions of the machine learning model to generate a meaningful output.

*Reliability patterns*

Store and restore online machine learning algorithm state.

**Summary of Streaming Data Processing Patterns**

This section has outlined streaming data processing patterns used by cloud native applications. [Table 6-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#streaming_data_processi-id00202) summarizes when we should and should not use these patterns, and their benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Transformation | To transform the event format, structure, or protocol. To add or remove partial data to or from the event. Third-party systems do not support the current event. | The consuming system has the ability to understand the event. | Allows incompatible systems to communicate with one another. Reduces event size by containing only relevant information. |
| Filters and Thresholds | Only a subset of events is relevant for processing. | All events are needed for decision making. | Reduces the load on the system by selecting only events that can produce the most value to the use case. |
| Windowed Aggregation | To aggregate events over time or length. To perform operations such as summation, minimum, maximum, average, standard deviation, and count on the events. | For operations that cannot be performed with fixed memory such as detecting the median of the events. High accuracy is needed without the use of reliability patterns. | Reduces the load on the system by aggregating events. Provides data summary to better understand the behavior as a whole. |
| Stream Join | To join events from two or more event streams. To collect events that were previously split to parallelize processing. | Joining events do not arrive in relatively close proximity. High accuracy is needed without the use of reliability patterns. | Allows events to be correlated. Enables synchronous processing of events. |
| Temporal Event Ordering | To detect the sequence of event occurrences. To detect the nonoccurrence of events. | Event sequencing cannot be defined as a finite-state machine. High accuracy is needed without the use of reliability patterns. Incoming events arrive out-of-order. | Allows detecting complex conditions based on event arrival order. |
| Machine Learner | To perform predictions in real time. To perform classification, clustering, or regression analysis on the events. | We cannot use a model to accurately predict the values. Historical data is not available for building machine learning models. | Automates decision making. Provides reasonable estimates. |
| Table 6-3. Streaming data processing patterns | | | |

**Scaling and Performance Optimization Patterns**

Cloud native applications that perform stream processing have unique scalability and performance requirements. For instance, these applications require event ordering to be maintained while processing events. Furthermore, as most of these applications have in-memory state, they also need a strategy to scale so they can process more events without compromising their accuracy.

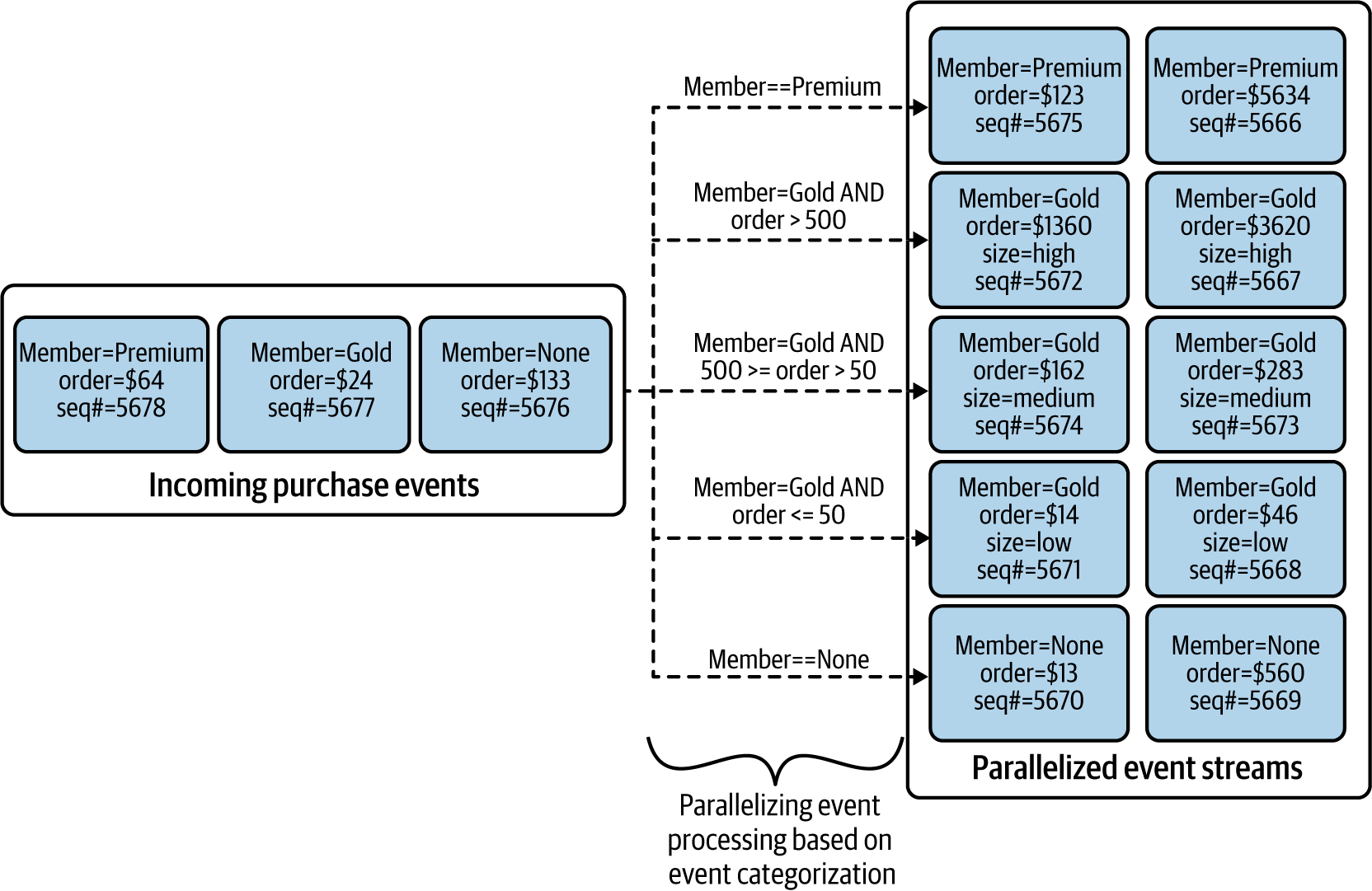
In this section, we discuss key patterns used to scale streaming cloud native applications. We also look at patterns commonly used to order events and to improve performance.

**Sequential Convoy Pattern**

The *Sequential Convoy pattern* scales cloud native stream-processing applications by separating events into various categories and processing them in parallel. It also works to persist event ordering so events can be combined at a later time, while preserving the original order of the events.

**How it works**

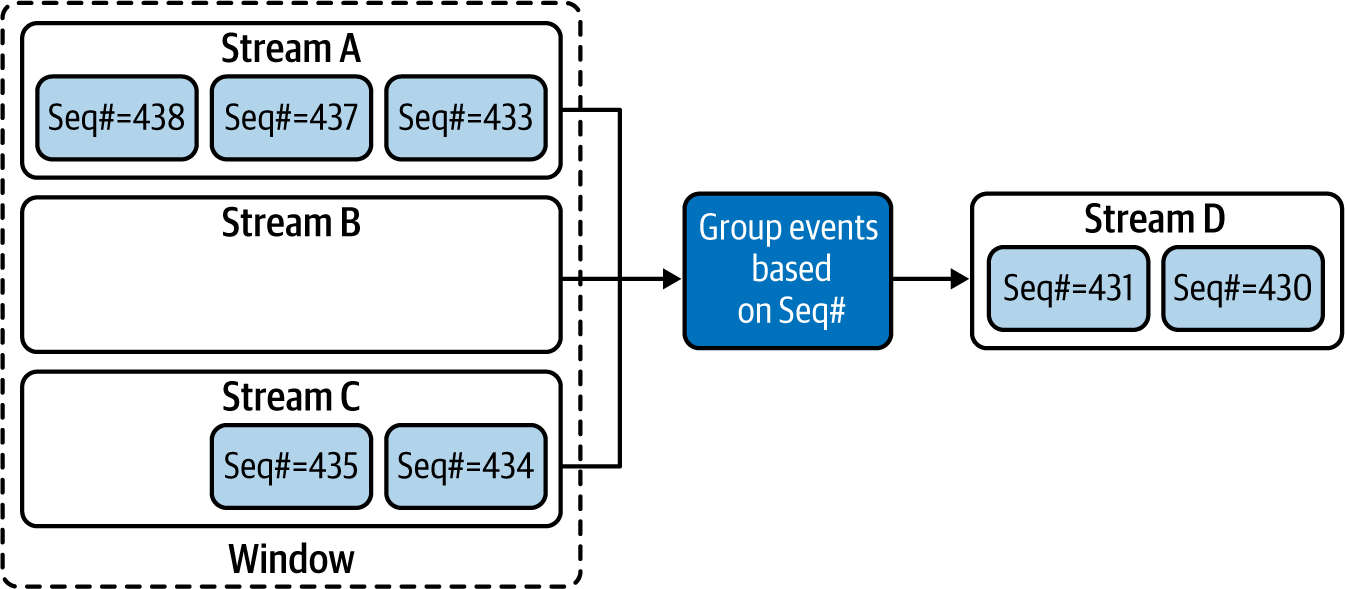
As the name suggests, this pattern sees events as items moving along a conveyor belt. It groups the events into categories based on their characteristics and processes them in parallel. One example is an ecommerce application that provides different product delivery time guarantees based on type of customer and order size ([Figure 6-13](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#partitioning_and_processing_order_event)). This application can categorize purchase events by the type of customer (such as premium, gold, or not a member) and can categorize gold member purchases by their order size (such as $50 or less as small, $50 to $500 as medium, and more than $500 as high). This allows events to be partitioned and processed in parallel to provide various types of delivery guarantees, such as the same day, in two days, and in one week.



**Figure 6-13. Partitioning and processing order events in parallel**

Each event also can be labeled with a sequence number before separation. This allows each substream to maintain the event order during processing by applications that require guaranteed ordering, such as when using patterns like Windowed Aggregations or Temporal Event Ordering.

This sequencing can also be used to join the parallel streams together at a later time, based on the original event order. We use a merge sort by selecting the smallest sequence number among all the substreams, as shown in [Figure 6-14](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#grouping_events_based_on_event_sequence). This enables us to group the events back in their original order and emit them for more processing.



**Figure 6-14. Grouping events based on event sequence number**

Some events are still stuck in the windows, as the event with Seq#432 has not yet arrived. Once that arrives, it will be emitted next, and the event with Seq#433 will follow. Likewise, other events will also be emitted in the sequence number order until the next missing sequence number is detected.

Message brokers and event queues, discussed in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns), play a vital role in realizing this pattern. They allow us to split and transfer events to substreams and to buffer events for grouping back into a single stream.

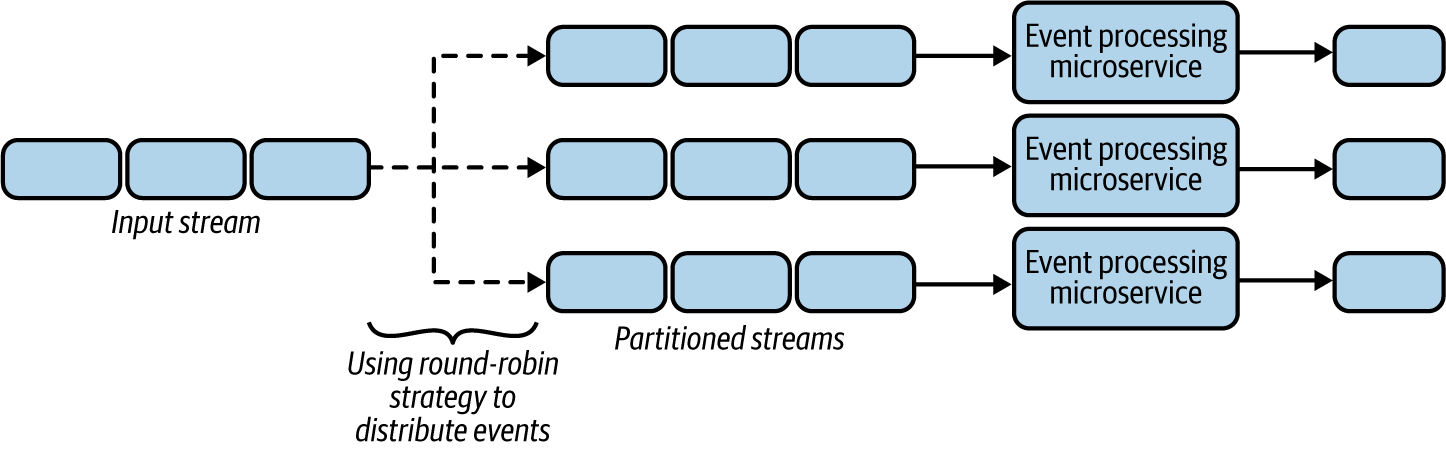
**How it’s used in practice**

This pattern is used for scaling event processing so we can process more events with cloud native applications that have limited memory capacity, and for partitioning events so that each substream is processed differently. Let’s look at how this pattern can be used in various scenarios.

**Scale stream-processing applications**

The Sequential Convoy pattern helps overcome cloud native stream-processing application limitations such as CPU, memory, and bandwidth, and allows us to process events with high throughput and low latency. For example, say we’re processing an event stream that transfers large amounts of confidential data. The events are encrypted and compressed to transfer the data much faster. Performing CPU-intensive operations such as uncompressing, decrypting, and transforming the data in real time is time-consuming, and performing such complex operations not only slows events, but also adds latency for the following events in the stream. This can lead to backlog buildup in the queues and cause bottlenecks for the whole system.

By simply partitioning the events into multiple substreams and parallelizing their processing, we can eliminate the latency addition and the event buildup in queues ([Figure 6-15](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#using_a_simple_round_robin_strategy_to)). We can use a simple round-robin strategy to distribute events to multiple substreams, which will process events much faster.



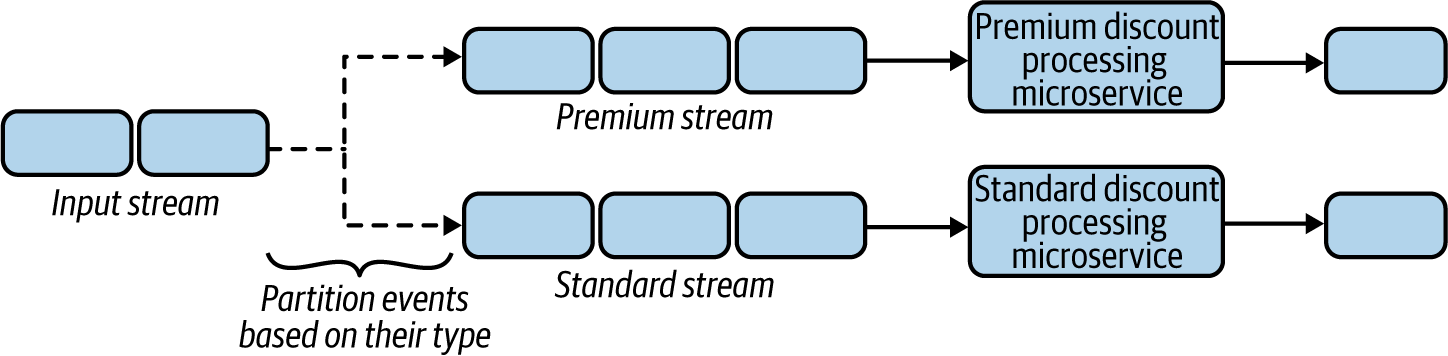
**Figure 6-15. Using a simple round-robin strategy to partition events**

Let’s say the event-processing microservice also needs to enrich the event based on customer ID from a data store lookup. Retrieving data from data stores can be time-consuming and can potentially introduce latency to the overall processing time. One way to improve the performance is by caching the data in the microservice, as discussed in [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns). But we cannot cache all the data because of the limited storage capacity of the microservice, and cache misses will still add latency.

We can use the Sequential Convoy pattern to separate events into different substreams based on hash values of customer IDs. This will process all events belonging to the same customer ID on the same node, improving the chance of cache hits. By reducing the number of customer IDs processed by a single microservice, we further increase cache hits, thereby meeting our performance goals.

**Partition the stream processing**

The Sequential Convoy pattern enables us to execute different use cases against the same event stream by partitioning different event types into parallel streams. For example, imagine an ecommerce platform providing extra discounts to premium customers. It can use customer data to categorize customers at different levels, such as premium and standard, and provide extra discounts for premium customers. By using the Sequential Convoy pattern, it can split the events into multiple substreams based on customer attributes, and then use different types of microservices to process those substreams through two pipelines to provide relevant discounts ([Figure 6-16](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#generating_different_pipelines_to_proce)).



**Figure 6-16. Generating different pipelines to process premium and standard events**

**Considerations**

The ability to categorize events based on event attributes is crucial for meaningful stream processing. Sometimes events can be categorized based on a single attribute such as customer ID or product ID, but in other cases we might need to combine various attributes such as order value or place of order to better process related events together and in parallel.

When events are separated in multiple substreams, they can go through various cloud native applications, and during this process some events get filtered and dropped. This can make the sequence numbers noncontiguous. Therefore, we have to be mindful when events are regrouped as we might not find all of them. We can assume events have been dropped based on the next emitted events. But when we cannot reliably determine missing events, we employ a time-out to determine that the events were dropped.

A better approach to regrouping events is an *end-of-sequence message*. This can be emitted by the processing applications with the last process message ID in a periodic time interval. This tells us that all IDs before the given end-of-sequence ID have been processed by the upstream applications and that the missing IDs smaller than the last ID are dropped messages. This unblocks the processing of later events.

When grouping based on sequence numbers is not possible, we can simply collect events and publish them to a single topic, and then use the Buffered Event Ordering pattern (discussed later in this chapter), to buffer and sort events based on sequence numbers or event timestamps.

It is also important to plan for scaling the number of streams when we detect that the stream-processing microservice is becoming a bottleneck. One approach is to reshard by altering the relevant categories to divide the saturated substream into multiple other substreams for processing events. We also need to migrate the application state across all substreams. Therefore, it is important to store the application state in such a way that we can separate those streams when we need to scale. We discuss in detail how to store state in [“Periodic Snapshot State Persistence Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#periodic_snapshot_state_persistence_pat).

**NOTE**

It is important to evaluate the throughput when using this pattern, and whether it meets expectations. For use cases that require extremely high throughput and low latency, adding sequence numbers to events and rejoining events based on those numbers can create a bottleneck, and in these cases we need to reevaluate whether ordering is really essential.

**Related patterns**

The following are related to the Sequential Convoy pattern:

*Producer-Consumer and Publisher-Subscriber patterns*

Can be used as the base for building the Sequential Convoy pattern. These patterns are covered in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

*Buffered Event Ordering pattern*

Provides an alternative way to order events while joining events from multiple event streams together. This pattern is covered next.

*Periodic Snapshot State Persistence pattern*

Stores substream application states and supports scalability. This pattern is covered later in this chapter.

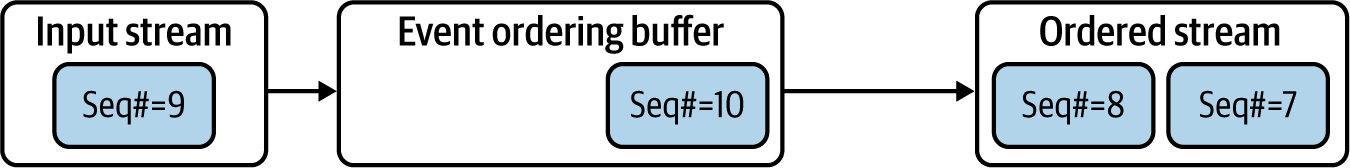
**Buffered Event Ordering Pattern**

Network delays and connection retries can cause events to get out of order. The *Buffered Event Ordering pattern* allows us to reorder events before processing them downstream. We can order events based on time or on the order they are generated.

**How it works**

For events to be ordered, they must have an incremental value by which to order them. This value can be a sequence number or a timestamp, for example. Sequence numbers will continuously increase, and we can guarantee that each event in a stream will have a unique number. But with a timestamp, we cannot guarantee that all events will have unique values, because multiple events can be generated in the same millisecond.

[Figure 6-17](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#ordering_events_based_on_sequence_numbe) illustrates the use of sequence numbers. If we have most recently received an event with sequence number 7, and now we receive an event with sequence number 8, we can immediately send it for processing because we know that 8 follows 7. But if after 8 we get 10, we know that we are missing an event and so cannot send 10 for processing. Instead, we need to use a time-out (of 30 seconds, for instance) to wait for the missing event. Then, if the missing event 9 arrives in time, we send it as well as event number 10. But if it does not arrive in time, we have to send the event with sequence number 10 before 9.



**Figure 6-17. Ordering events based on sequence number**

The event with sequence number 9 could be dropped upstream for many reasons. Rather than waiting for the time-out, the previous processor can send an empty event with sequence number 9. We could also annotate the next event, number 10, stating that it has processed all events before that. This can help the ordering application make a faster decision on whether an event has been dropped, without adding latency.

When ordering events by timestamp, we cannot perform this type of optimization, as we are working with duplicate and missing timestamp values. We need to rely on the last-seen timestamp and wait for time-outs to order the events.

When we detect an out-of-order of event, through either timestamps or sequence numbers, we need to modify the wait time-out so that this issue is less likely to occur. For example, if we receive an event with a timestamp of 700 seconds and then receive an event with a timestamp of 650 seconds, we need to increase the wait time-out by the event time gap—50 seconds. For more details on implementing this behavior, refer to algorithms such as K-slack and AQ-K slack.

**How it’s used in practice**

This pattern can be deployed in front of any use case that needs ordered events, as long as the events have attributes that can be used for ordering.

**Order events generated on distributed event sources**

Events generated by distributed sources usually become out of order because of data transmission latency added by network and intermediate systems. Consider an example of distributed surveillance sensors emitting events. These events can reach the processing system at various times because of transmission latency, so they will be out of order when we combine all of them into a single stream for processing.

As the sensors are distributed, the only way to order the events is through timestamps. The out-of-order events can be sent to a single topic in a message broker, and by using a microservice, those events can be fetched, reordered through the Buffered Event Ordering pattern, and sent downstream for further processing. We can use this pattern only when the sensors have their times synced and the ordering based on the timestamps is reasonably accurate for processing.

**Reorder events generated from the same event sources**

Often we need to parallelize event processing to achieve performance, and then reorder events into their original sequence for further processing. For example, say we want to parallel-process user interaction on a browser and merge those events back in order.

Because all the events needed to track user behavior are generated from the same browser, we recommend adding sequence numbers to those events along with the timestamp. This will not only allow us to process the events in parallel, to improve efficiency, but also group them back together, as we discussed in the Sequential Convoy pattern. We can feed the processed events to a topic in a message broker, and use a cloud native application to fetch and order the events based on their sequence numbers.

**Considerations**

This pattern is useful when we need to aggregate events over time or when a sequence of actions needs to be detected. In all other cases, we do not recommend using this pattern, as it can add latency or introduce a bottleneck to the system.

Events can be reordered with high accuracy, but only when those events are generated from a single source. We can’t ensure that ordering by event timestamps will produce true ordering, as the distributed sources that generate events will not have their timestamps synchronized to the millisecond.

When events are generated from a single source, always try to add sequence numbers to the events along with the timestamp, because ordering events based on sequence number is much more efficient. It also does not add the same amount of latency as ordering events by timestamp.

When we have detected a late-arriving event, we have to decide whether to send the out-of-order event forward for processing or drop it. This decision depends on the use case. For example, if the events are reporting a current status (like temperature of the furnace in an industrial setup), dropping an old event will not cause issues because we have more-recent data for processing. But if the events are credit card transactions that we track to monitor fraud, dropping events can cause issues. This can lead to detecting invalid sequences if the processing application is using patterns such as Temporal Event Ordering.

The microservice implementing the Buffered Event Ordering pattern needs to store some events in the buffer while it is waiting for older events to arrive; this means that the microservice has state. It is important to employ reliability patterns like Periodic Snapshot State Persistence or Replay so the service can recover its state across failures and restarts.

**Related patterns**

The following patterns, covered in this chapter, are related to the Buffered Event Ordering pattern:

*Temporal Event Ordering and Windowed Aggregation patterns*

These patterns can benefit from the Buffered Event Ordering pattern, as they require events to be ordered to produce more-accurate results.

*Reliability patterns*

For storing and retrieving events that are waiting in the buffer for ordering, during system failure and restart.

**Course Correction Pattern**

The *Course Correction pattern* attempts to report its analysis of events as soon as possible, and then later correct its analysis and report again, as soon as it retrieves missing (or late) events. This produces early analysis with low latency rather than sending an accurate analysis with higher latency.

**How it works**

This pattern should be combined with patterns like Windowed Aggregation or Temporal Event Ordering. Rather than waiting for all events to arrive, we send aggregation and event sequence detection as soon as we have a result. The results of the aggregation and sequence detection are an early estimate and may not be accurate. Later, when we receive missing events, we send updated results.

**NOTE**

For this pattern to work, downstream applications should be able to know that these events can be partial updates and to adapt based on more-accurate updates that will arrive later.

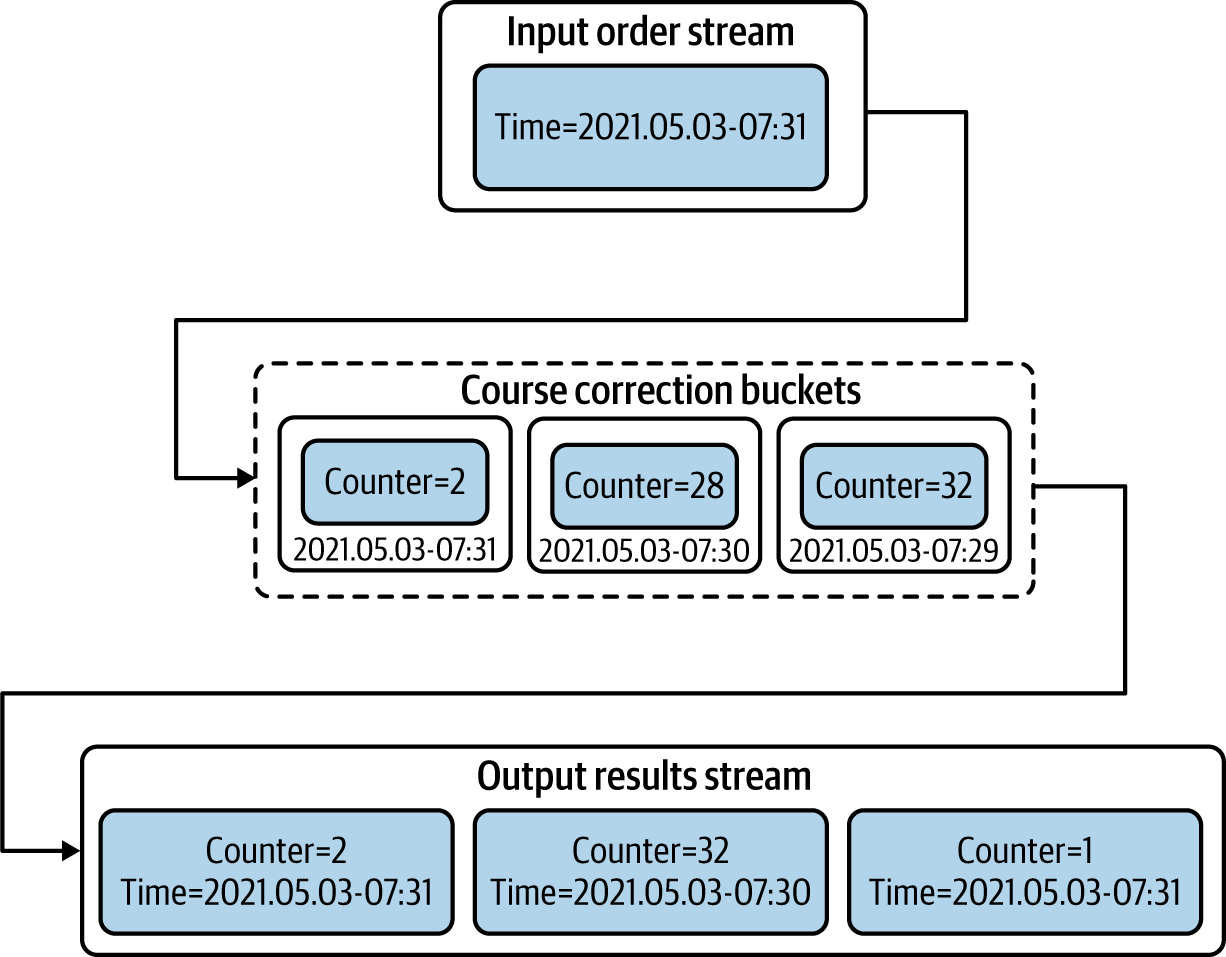
**How it’s used in practice**

This pattern should be used only when we need events in order, have a requirement for low latency, and can cope with inaccurate early estimates. Let’s consider some example scenarios to understand this in more detail.

**Update results with new information**

This pattern is commonly used when users are eager to obtain aggregated results quickly. This can especially be useful when we are displaying results in real time on a screen. In these cases, we simply need to hold the events for more time, and alter the decision based on late event arrivals.

For example, let’s say we want our application to calculate the sum of orders arrived per minute, and report the number of orders on a per-minute basis. As shown in [Figure 6-18](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#aggregate_events_over_multiple_buckets), we simply need to create buckets for each time period, such as 2021.05.03-07:30 and 2021.05.03-07:31, denoting the time periods in minutes, and then keep a counter within the bucket to continuously count the values arriving during that period. We will be able to emit the events when the time period ends, as well as send an update if events arrive later, such as sending an update for 2021.05.03-07:30 along with the output of 2021.05.03-07:31.



**Figure 6-18. Aggregate events over multiple buckets to course-correct previous results**

Be careful about when to purge the buckets, as having multiple buckets can use large amounts of memory. Purging them early can cause calculation errors, as events arriving late won’t have their respective bucket with previous aggregations.

**Correct previous decisions**

Sometimes we need to make an early decision, and when the situation changes, we send compensation events so that corrective actions can be taken. Let’s say we want to dispatch a taxi as soon as a user requests one.

When a user requests a taxi, we broadcast that message to all taxis in the region, and when we know a taxi has accepted the ride, we send another broadcast message to all taxis to inform them that the ride is accepted. During the initial request, there is a chance of multiple taxis accepting it, but we discover that only later because of network delays. Because we could mistakenly dispatch more than one taxi for the ride, we send a correction event to cancel the other taxi assignments.

**Considerations**

This pattern can be used only when early estimates are useful and the use case allows for compensation or course correction based on an update. For use cases that do not support course correction, we have to delay the decision making by using patterns like Buffered Event Ordering.

In most cases, events will be stored in memory while we are waiting for late event arrivals. This can cause high memory usage, so we need to find a balance between how long the system can wait for late events without running out of memory.

Since course correction also needs to remember previous values or previously emitted results, we need to apply reliability patterns to ensure that their state is preserved across system failures and restarts.

**Related patterns**

The following patterns, covered in this chapter, are related to the Course Correction pattern:

*Reliability patterns*

For storing application state holding previous events and previously emitted outputs.

*Buffered Event Ordering pattern*

Can be used instead of this pattern when the use case does not support course correction.

*Temporal Event Ordering and Windowed Aggregation patterns*

These patterns can benefit from the Course Correction pattern as they can use course correction to correct their early estimates.

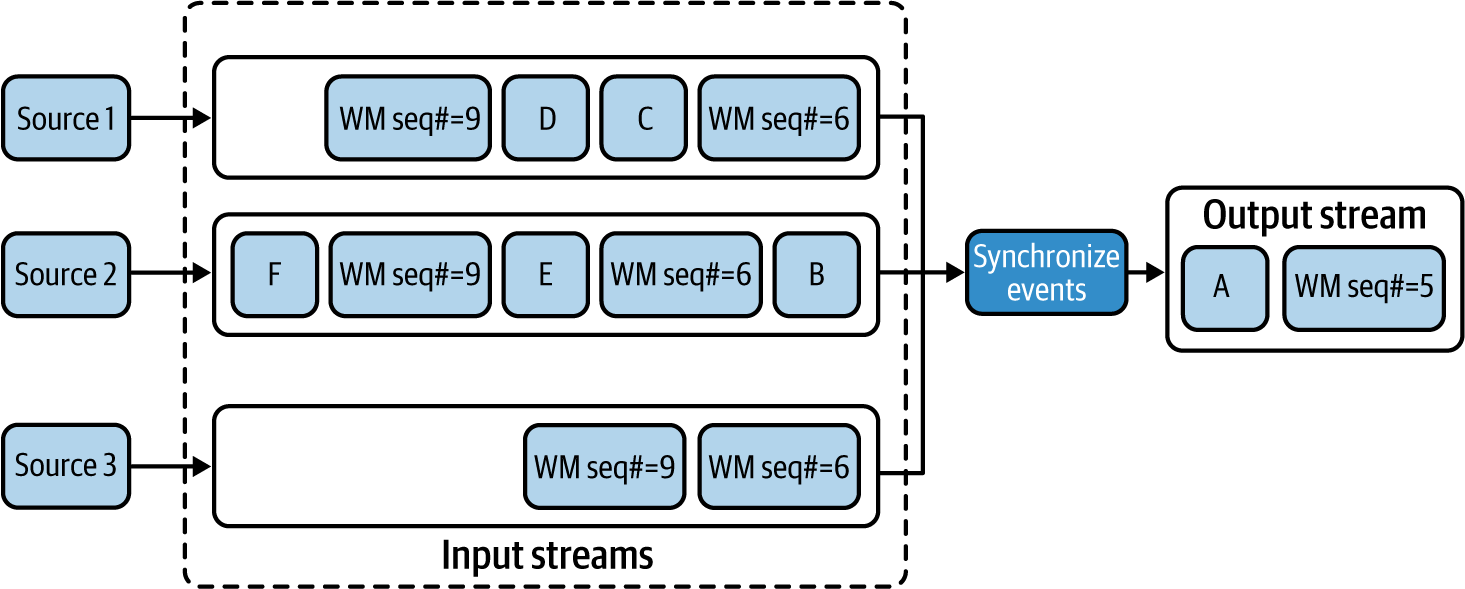
**Watermark Pattern**

The *Watermark pattern* is useful for periodically aligning stream processing across multiple microservices within a cloud native application that are connected in a mesh-like structure via event streams. This alignment will help determine whether all microservices have processed all arrived events before a given event, which is commonly referred to as the *watermark event*. We can use this pattern to sync multiple microservices without using system times.

**How it works**

For watermarks to work, a watermark generator should generate a watermark event periodically and send it through all the external inputs of the cloud native application. This event should be considered special, and microservices should pass it through to their dependent systems. We also need to be sure that each intermediate microservice that consumes this event can resend it in the same position among the sequence of events it has received and processed, and not before or after other events.

When the input systems are time synchronized, we can make those systems independently generate the watermark events at given intervals, such as once every minute or every five minutes, as shown in [Figure 6-19](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#generating_watermark_events_and_synchro).



**Figure 6-19. Generating watermark events and synchronizing events based on them**

When the microservice receives a watermark event in a stream (such as the watermark event with sequence number 6 in this example), it should not continue processing any more events from that stream, and process only events from other streams (such as Event B of the second stream), that have not yet received the corresponding watermark event. When we receive all corresponding watermark events on all streams, we can pass that watermark event to all its dependents and continue processing other events from all the input streams until we receive the next watermark event in a stream. This process is repeated, and this approach ensures that event processing is synchronized at each watermark event.

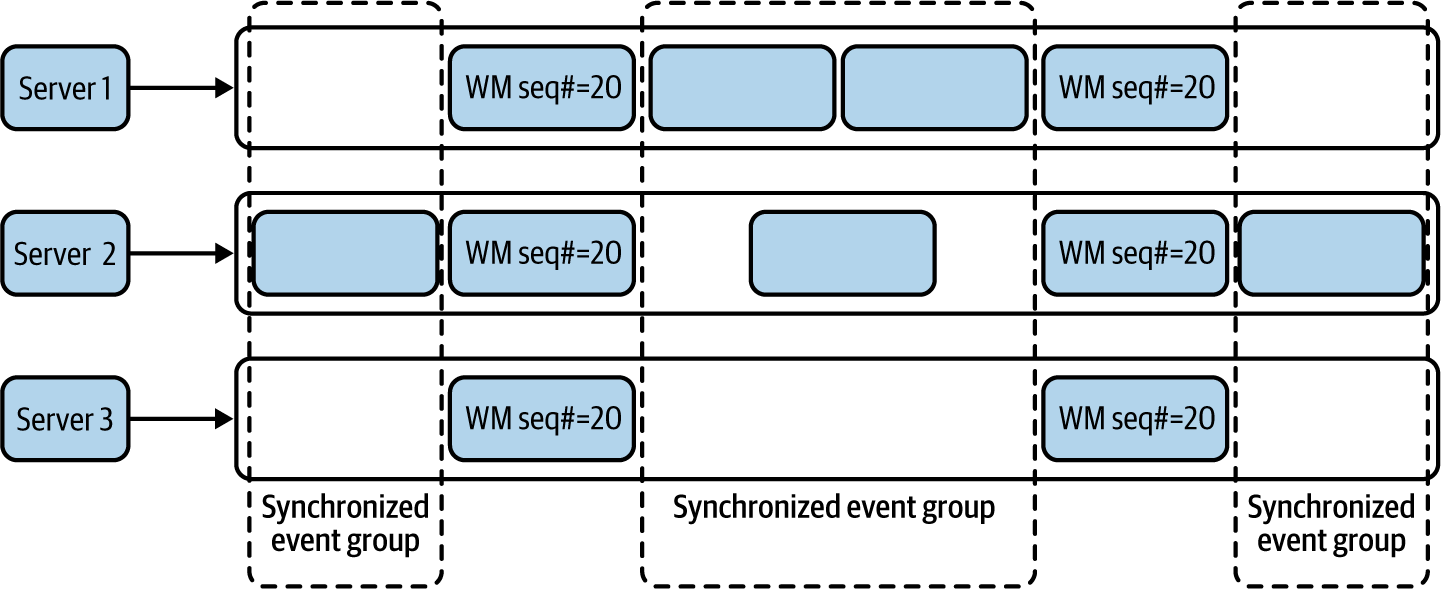
When the preceding options are not possible, we can also make the input sources poll a global counter to fetch the next watermark event sequence number and emit it periodically along with the events. In this case, we should make sure that watermark events arriving in multiple streams are processed in a sequential manner. If we find a sequence number out of sync, we should halt the execution of events from that stream until we receive a watermark event with a lower sequence number on another stream.

**How it’s used in practice**

This pattern can be used when multiple source systems are not synchronized on time, or when network latency or processing time can affect event arrival time. In this case, events in one stream can arrive earlier, while events from other streams can arrive later, and this can cause issues when analyzing events across multiple streams. This pattern is ideal for synchronizing the event processing periodically to reduce errors.

**Synchronize events generated from event sources that are time synchronized**

Watermarks can be used to generate synchronized event groups that can produce accurate aggregation results. Consider aggregate readings from multiple servers in a server farm that are in sync and emit events. We simply need to emit the watermark events at given time intervals to the input streams generated by those servers. This helps us collect the events between those watermarks and perform aggregation operations, as shown in [Figure 6-20](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#aggregating_events_based_on_synchronize). This also ensures that the given aggregation is accurate and not affected by any network delays or other external factors.



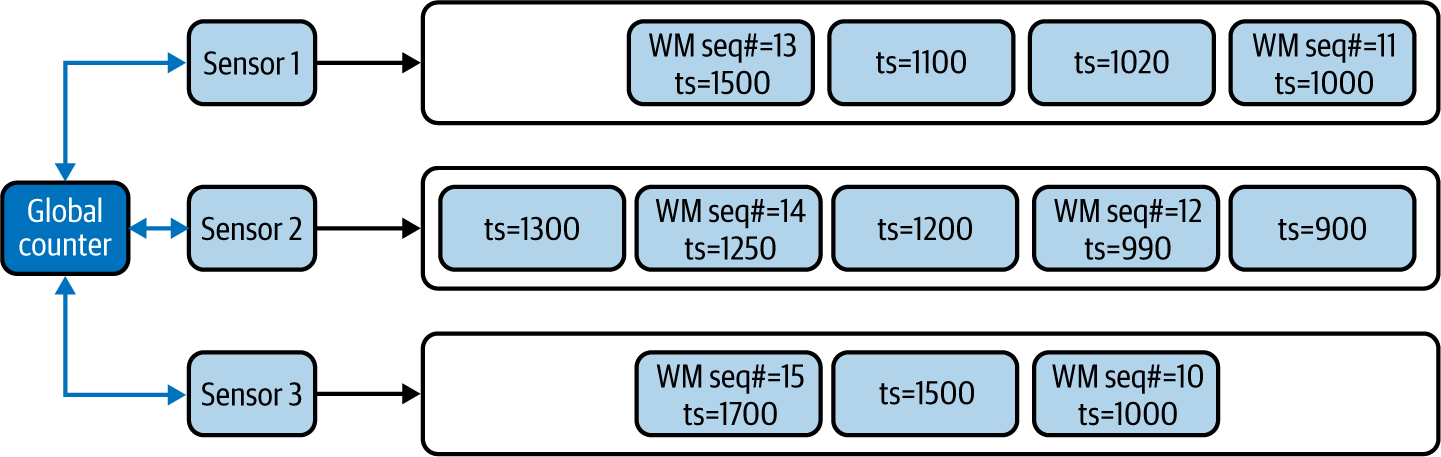
**Figure 6-20. Aggregating events based on synchronized event groups generated between watermark events**

**NOTE**

In this case, we are not using event timestamps for synchronization because some streams may not publish events for a long period of time.

**Synchronize events generated from nonsynchronized sources**

Let’s say we want to detect interesting incidents by using Temporal Event Ordering from multiple surveillance sensors deployed across the neighborhood. As events are emitted from distributed sensors, some sensors can be emitting them with a delay, or their events may arrive later because of network latency. To enforce synchronization, each sensor client can periodically fetch a sequence number from the global counter deployed on a central server and inject it along with the sensor reading. [Figure 6-21](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#ordering_events_based_on_continuous_wat) shows that we can synchronize the events based on the sequence number of the watermark events, and use the timestamps of those watermark events to determine the relative time of events, thereby determining the true order of the events and detecting event occurrence patterns.



**Figure 6-21. Ordering events based on continuous watermark events**

**Considerations**

This pattern is useful only when we know that significant differences exist in event arrival times because of network latency or when the input systems do not have their times in sync. In other cases, this pattern does not bring us many advantages. For example, when systems are already synchronized on time, we can use the Buffered Event Ordering pattern to sort events by their timestamps.

**NOTE**

This pattern can only *reduce* the time synchronization issues among streams; we cannot guarantee that it can produce accurate results for all use cases. Even with periodic time syncs, we can have issues resolving which event of one stream arrived before an event from another stream.

We also do not recommend using this pattern unless you have a strong reason to process events on the actual time they are generated and need relative ordering of those events. Avoid using this pattern when it is not truly necessary because of the architectural and technological complexity it adds to the whole infrastructure.

Where possible, we also recommend using stream-processing systems like Apache Flink, which has the watermarking feature available by default. Using a stream-processing system is beneficial when your use case depends heavily on the order of events, such as fraud detection or surveillance.

**Related patterns**

The following are related to the Watermark pattern; all are covered in this chapter:

*Buffered Event Ordering and Course Correction patterns*

Can be used instead of this pattern when event arrival times are not affected by network latency or other processing delays by the systems.

*Temporal Event Ordering and Windowed Aggregation patterns*

These patterns can benefit from the Watermark pattern as they require events to be ordered to produce correct results.

*Periodic Snapshot State Persistence pattern*

The Watermark pattern can be a prerequisite for Periodic Snapshot State Persistence to perform state snapshots in a synchronized manner among multiple streams.

**Summary of Scaling and Performance Optimization Patterns**

In this section, we outlined scaling and performance optimization patterns often used by cloud native stream-processing applications. [Table 6-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#scaling_and_performance_optimization_pa) summarizes when we should and should not use these patterns and their benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Sequential Convoy | To scale stream-processing applications. To partition streams so each stream can be used for various use cases. To allow processing events in parallel and regroup them based on the original order. | Streaming applications have enough capacity to process the events. | Supports scalability of stream processing Preserves event ordering when events are processed in parallel. |
| Buffered Event Ordering | To order events based on timestamp or sequence number. To order events that are already out of order and published via a single event stream. | To group events from multiple ordered event streams. We need true ordering of events that are generated from distributed sources. Reliability patterns cannot be applied to the application. | Can be applied in front of any application that needs events in order. |
| Course Correction | To correct previously produced results. To produce early aggregation estimates. To guess the event-sequence order and correct the decision later. | The dependent downstream applications cannot handle continuous event updates. | Allows us to produce early estimates and correct them as we have more data. |
| Watermark | To perform aggregation operations on event streams that are out of sync. Try to order events that are generated by distributed systems. | We cannot inject watermark events closer to the event sources. Intermediate systems cannot bypass watermark events. Network bandwidth is a concern. | Periodically synchronizes events across multiple streams. Helps overcome network and processing latency added by intermediate systems. |
| Table 6-4. Scaling and performance optimization patterns | | | |

**Reliability Patterns**

Most cloud native stream-processing applications store their state in memory so they can process events with low latency and high throughput. For this reason, reliability is key for stream processing applications. As we’ve mentioned throughout the chapter, *reliability patterns* guarantee that the state of our applications can be preserved across failures and restarts.

Reliability patterns also help us achieve at-least-once event delivery—sending an event one or multiple times—and allow exactly once event processing—processing events once and only once, even during system failures. This section dives deep into various reliability patterns that are used to preserve cloud native application state during critical situations.

**Replay Pattern**

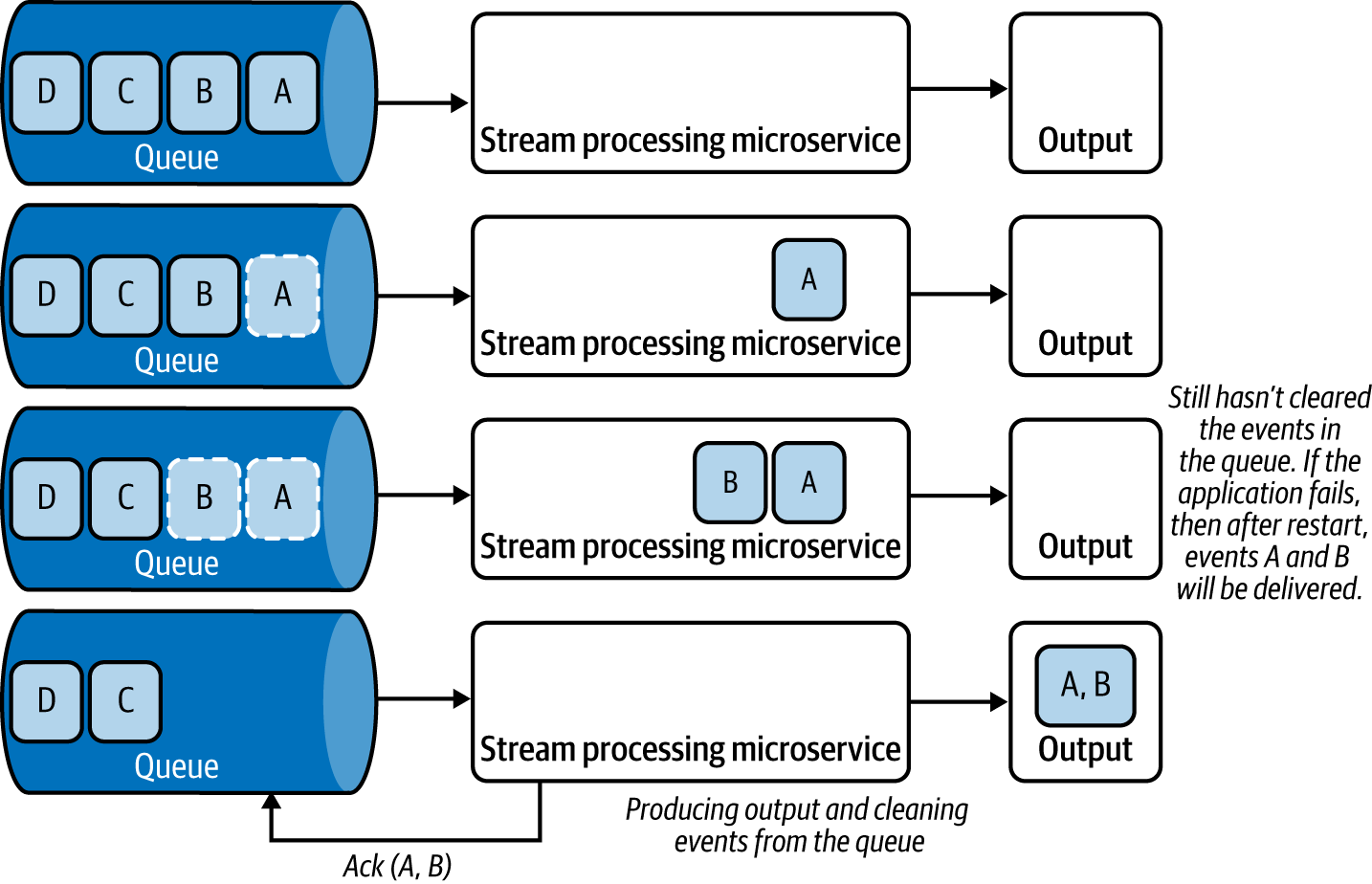
By using the *Replay pattern*, the state of a microservice can be restored by replaying the events it has processed in the past, especially when its state depends only on recent events.

**How it works**

This pattern works by resending events when the system is down. The number of old events it needs to send depends on the use case. For example, if the microservice is aggregating events over the past three minutes, then during failures, resending the events arrived during the last three minutes is sufficient.

When the stateful microservice can store its state periodically, we will be able to identify the last successfully processed event from the latest snapshot, and we should be able to replay all events arrived after that.

To re-create the state of a system, the source of the data should contain the events even after they are retrieved by the microservice. We cannot use standard message brokers with their automatic event acknowledgment feature, because the events will be deleted from the message broker as we read them, unless we use queues or durable subscriptions and differ the acknowledgment of consumed events. As shown in [Figure 6-22](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#deferring_acknowledgment_until_after_ou), we can delay sending acknowledgments to the queues until the microservice is done processing and cleaning out its state, or until it persists its state in durable storage.



**Figure 6-22. Deferring acknowledgment until after outputs are generated**

We can use this pattern with microservices that consume events from log-based message brokers, such as Kafka and NATS, because these brokers will not delete the events when they are delivered to the microservice, and the microservices can request events to be played back from the last sequence number they have successfully processed. We can also use this pattern when events are read from persistent data stores like RDBMS databases, NoSQL stores, or filesystems.

**How it’s used in practice**

This pattern can be used to restore an application’s state by replaying the lost events due to system failure or restart. Let’s look at how this pattern could be used in a few scenarios.

**Replay events when system state is not persisted**

What if we want to generate one-minute aggregations of purchase orders? We are processing data in one-minute batches, and at the end of each minute, the microservice generates the aggregation and clears its state. If we are retrieving events from a durable topic subscription, we can defer the acknowledgment for the retrieved events until the end of the minute; this will ensure that we are deleting the events from the message broker only when we have successfully processed them and produced the output.

Let’s consider another example—analyzing log files for errors—and let’s assume that the event processing is stateless. We can model the microservice in such a way that it will move the log file from one folder to another when it is done processing that file. Then, if the application fails during the processing, and when it is restarted or when a new instance of the service is spawned, it can reprocess the last file as it has not yet been moved to the done folder. This ensures that all events are processed.

**Replay events when the system persists its state**

What if we want to aggregate the average temperature over the last hour? Let’s assume the processing microservice consumes events from Kafka and sends updates every minute to its dependents. By storing its aggregation state to a data store every minute when sending the updates, during failure, it can retrieve the last state from the data store, and replay all events from that point.

**Considerations**

Though the Replay pattern helps re-create the application state, it can produce duplicate events as a result of the replay. In this case, the dependent systems should be idempotent. We should not use this pattern when duplicate events can cause confusion on the part of dependent systems.

Sometimes events get lost even with the Replay pattern. For example, say we buffer events at the source for an extra two minutes so we can republish on demand. If the processing application encounters a failure, and it takes more than two minutes to start up, the source might have dropped events to accommodate new events. Therefore, we recommend using this pattern when consuming from event sources that can store events for a longer time period. We also recommend using this pattern in conjunction with the Periodic Snapshot State Persistence pattern, which we discuss next, especially when the processing application needs to persist state.

**Related patterns**

The following are related to the Replay pattern:

*Publisher-Subscriber pattern*

Can be used to establish durable subscriptions with event sources so they can be replayed during failure. This pattern is covered in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

*Periodic Snapshot State Persistence pattern*

This can be used in conjunction with the Replay pattern to restore application state and reduce the time to bring the application back alive. This pattern is covered next.

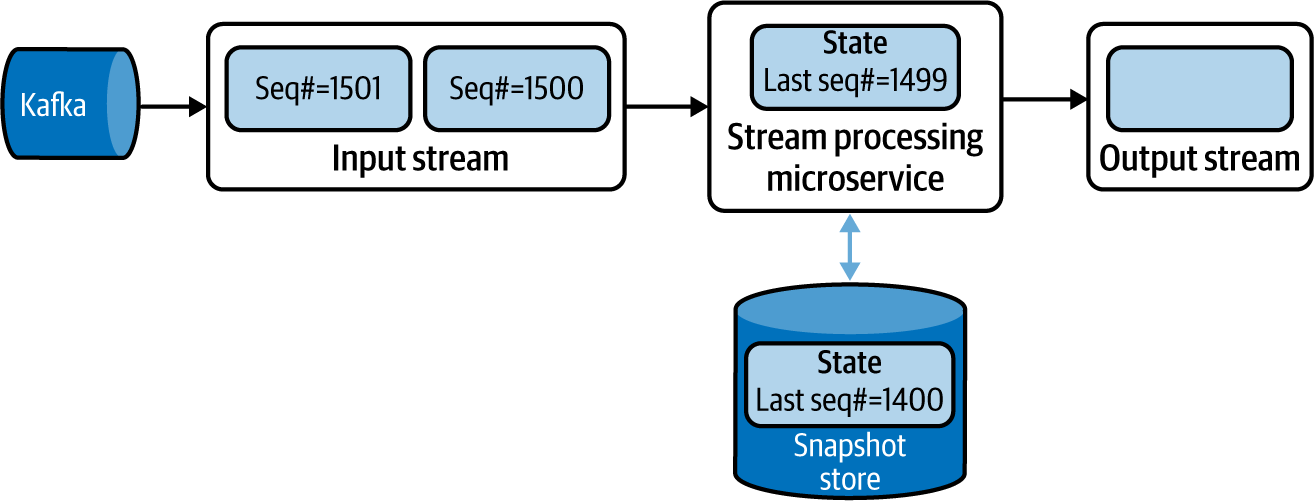
**Periodic Snapshot State Persistence Pattern**

Persisting the application state upon processing each incoming event is not feasible, as this introduces extremely high latency to cloud native applications due to the round-trip time of accessing state. The *Periodic Snapshot State Persistence pattern* allows us to persist the application state in a periodic manner so that we can restore the state reliably after system restarts or failures.

**How it works**

This pattern periodically makes a copy of its current state and persists that to a durable store between processing events. For this to work, we should ensure that the microservices can read and write state to a persistent storage (for example, Amazon S3).

To ensure that events are not lost during failures and to guarantee at-least-once event delivery, we must use message brokers to retrieve events. When using a log-based message broker like Kafka, we should store the event sequence number with the snapshot ([Figure 6-23](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#storing_periodic_state_snapshots_with_k)). With this approach, upon a restart, we reload the last stored snapshot and request the message broker to deliver events from the stored event sequence number.

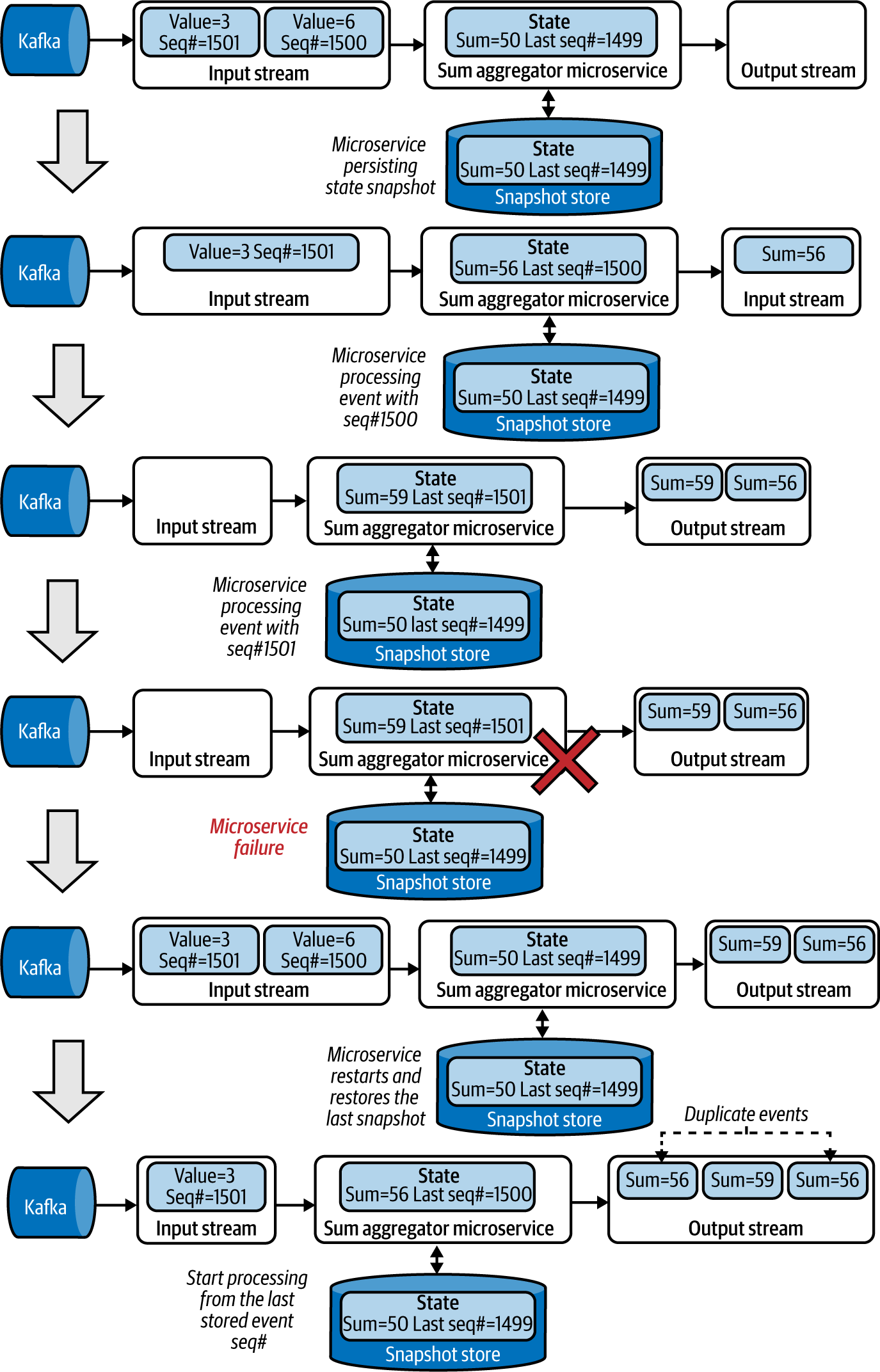


**Figure 6-23. Storing periodic state snapshots with Kafka event sequence numbers**

When using standard message brokers like ActiveMQ, we should acknowledge the processed messages only when storing the snapshot. This way, we can ensure that when the microservice is restarted the message broker sends all unacknowledged events.

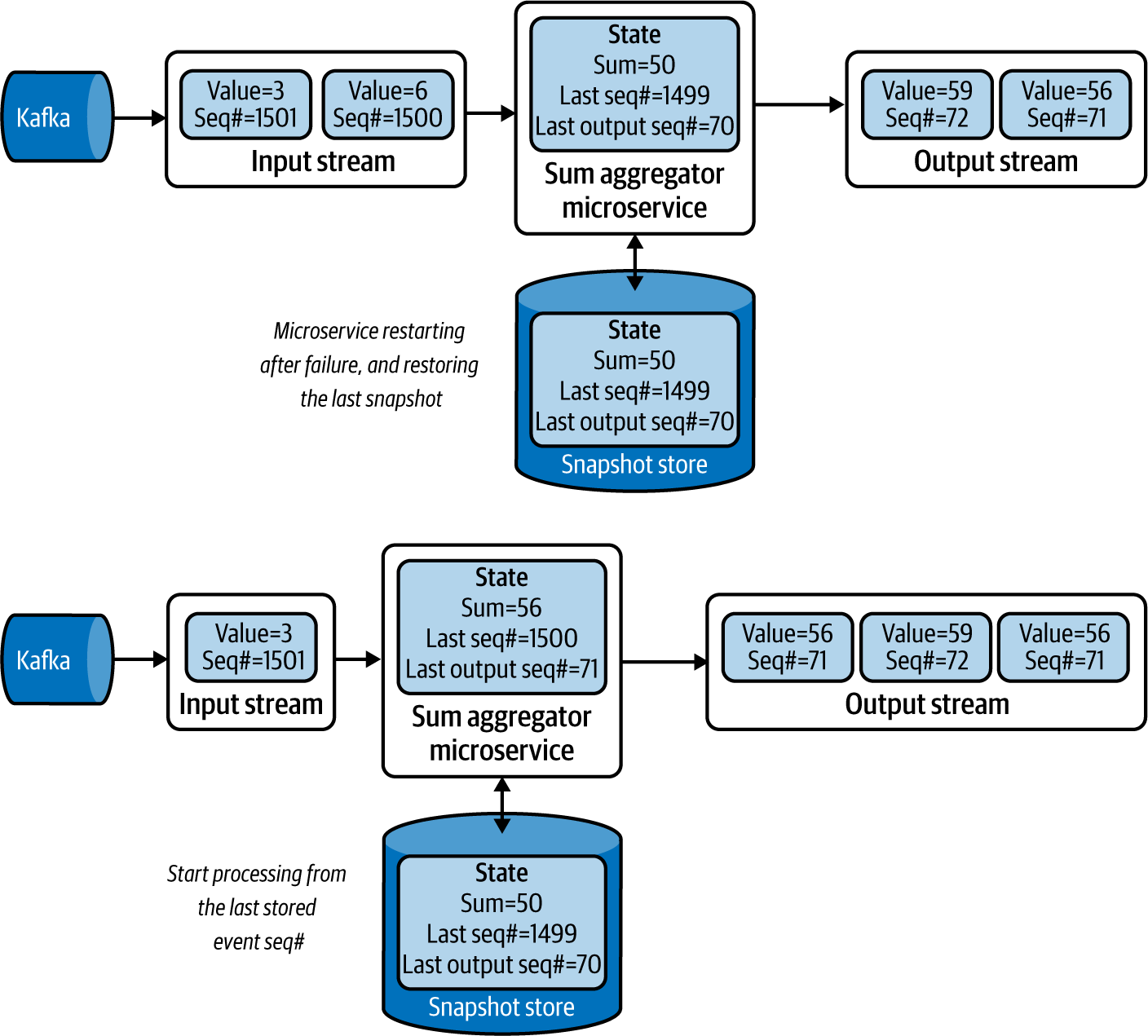
By using these approaches, we can ensure at-least-once delivery. In this case, the events arrived after the last snapshot, and the failure will be sent again. Therefore, those events will be processed again and sent to the downstream systems a second time, causing duplicate event delivery.

[Figure 6-24](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#performing_an_aggregation_operation_wit) shows an example of consuming events from Kafka and processing an aggregation operation The microservice performs a snapshot after processing an event with sequence number 1499 and when the running sum is at 50. Then it processes two events with sequence numbers 1500 and 1501, producing output events with sums 56 and 59, respectively, before failing. After restart, the microservice restores its state from the last snapshot, requesting Kafka to publish events that arrived after event 1499. This leads Kafka to send the events with sequence numbers 1500 and 1501 again, causing the microservice to reprocess and output events with sums 56 and 59, respectively, and duplicate event delivery to downstream systems.



**Figure 6-24. Performing an aggregation operation with snapshot state persistence and restoring state after system failure**

If the dependent systems do not support duplicate events, we can also introduce a new output sequence number from the microservice ([Figure 6-25](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#microservice_emitting_the_output_sequen)). The dependent systems can detect that the same sequence is repeating and drop the duplicate events without reprocessing them. This will achieve exactly once event processing even across system failures.



**Figure 6-25. Microservice emitting the output sequence number for the dependent system to detect duplicate events**

Instead of taking snapshots at periodic intervals, this pattern can also leverage the Watermark pattern to take snapshots only when event synchronizations are based on watermark events. This will ensure that all events before the given watermark are processed and persisted.

**How it’s used in practice**

This pattern can be used to store state when microservices process data in memory, and when their state cannot be persisted after every event.

Let’s say we want our application to detect if a stock price has continuously risen over the last 10 minutes. We need to keep track of only the last time we saw a dip in the stock price, and continuously check if that time is now older than 10 minutes. If so, we will alert the user.

If we are retrieving the events from a Kafka topic, we also need to remember the last-processed event’s sequence number along with the current stock price, and the last time we saw the stock price dip. These three represent the *state* of the microservice. To recover the microservice from failure, we need to persist all three values to a database or similar storage, in a periodic manner. During the recovery process, the microservice can restore the last stored snapshot and replay the Kafka events from the last stored event sequence number to ensure that the system state is preserved across system failures.

**Considerations**

We should use this pattern only when we are processing critical data that cannot be lost on system failure, because the pattern introduces significant operational overhead that is not worthwhile if data loss is acceptable. For example, if our processing window is small (say, one minute), system failure impacts for only as long as the system is down. To restore state, we can use the Replay pattern to reprocess the events during the lost period. But if our application state contains data from the previous day, we need to replay events from the previous day to re-create the state, which may not be feasible because of the processing time for the quantity of events. In such cases, we advise using the Periodic Snapshot State Persistence pattern.

In some situations, the state itself is quite large and requires significant time to store and retrieve. To mitigate this, use incremental snapshots, store only the delta between the current state and the last snapshot, and then replay incremental snapshots to re-create system state. For example, when using a time window of five minutes with one-minute time shifts for aggregation, store snapshots every minute with only the changes that happened during the last minute. When there is a failure, we load the last five snapshots to re-create the state of the five-minute window.

**NOTE**

When performing snapshots and persisting them in a data store, we recommend using threads, or something similar, so persisting state doesn’t block the processing of more events.

We do not recommend making the snapshot interval overly short, as this introduces overhead without significant benefit. But don’t set the snapshot overly long either, as this leads to not only writing and reading bigger snapshots (which takes longer), but also replaying more events on application restoration (which can increase the time for applications to become live again).

**Related patterns**

The following patterns, covered in this chapter, are related to the Periodic Snapshot State Persistence pattern:

*Temporal Event Ordering and Windowed Aggregation patterns*

These patterns can benefit from the Periodic Snapshot State Persistence pattern, as they require state to be stored to achieve reliability.

*Replay pattern*

This is used to re-create states from the missing events based on the last snapshot loaded during application recovery.

*Watermark pattern*

This can be used to synchronize state snapshots across multiple microservices.

**Two-Node Failover Pattern**

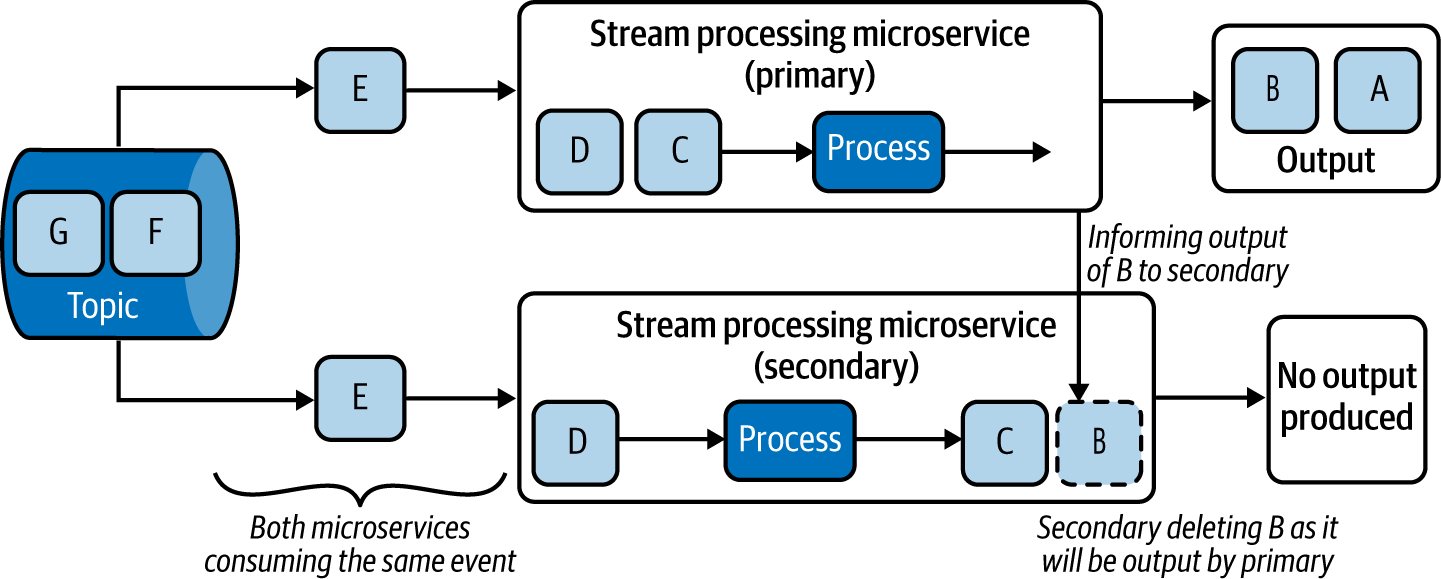
Low-latency microservices do not have the luxury of taking a couple of minutes after failure to restart and restore their states. For these microservices, it is operationally superior to run a redundant microservice to allow failover. We can run such a microservice by using the *Two-Node Failover pattern*.

**How it works**

This pattern focuses on running a parallel backup microservice. When microservices are deployed, they perform a leader election; we can use systems such as ZooKeeper or native cloud services to designate one microservice as primary and the other as secondary.

These two microservices, shown in [Figure 6-26](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#running_microservices_as_primary_and_se), will process all the incoming events. Both microservices consuming the same event are achieved by subscribing to a common topic, as discussed in the Publisher-Subscriber pattern in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns). As both microservices process the same events, they have consistent state. Only the primary will be emitting the output to its dependencies, and it also sends the output to the secondary. The secondary matches its output against the primary’s and drops all events that have been processed by the primary (Event B).

At this point, if the primary fails, the secondary will be promoted to become the primary, and, while continuing to process events, it will also start publishing to the dependencies. Because it knows the last event published by the previous primary (Event B), it will be able to determine which events have not yet been sent and send those events (Event C) first before publishing new events (Event D).



**Figure 6-26. Running microservices as primary and secondary to enable failover**

If the failed microservice is restarted in a later state, it is demoted to operating as the secondary, where it will consume events, process them, and drop them based on the outputs generated by the current primary.

Over time, the secondary’s state will become equal to its primary and will become eligible to be a failover target. If the microservices leverage Periodic Snapshot State Persistence, they can recover quickly by restoring state faster and improving the availability of the system.

**How it’s used in practice**

This pattern can be used when degradation of latency and system downtime cannot be tolerated. Say we retrieve stock order events from NATS, process the number of stock bids and asks in real time, and publish them to stockbrokers so they can instantly identify changes in market trends. Using the Two-Node Failover pattern, we cannot only switch to the secondary node instantly upon detecting that the primary has failed, but also ensure that no events are dropped. This is because the secondary publishes only data that the primary has not.

**Considerations**

Use this pattern only when low latency is the main requirement. Otherwise, use other patterns such as Periodic Snapshot State Persistence. This pattern is complex to implement, and the architectural complexity is not worthwhile if we can tolerate downtime during system failure.

This pattern requires both microservices to have robust connectivity, as the primary needs to publish its output to the secondary. Furthermore, a risk of network partitioning between the primary and secondary exists. In this case, we require a third system to function as the leadership elector; otherwise, both microservices could become primary in parallel and send outputs downstream.

We should also be mindful that both microservices can fail simultaneously; then the system would become unavailable and we’d lose their state. As a mitigation, we can adopt the Periodic Snapshot State Persistence pattern or Replay pattern to allow for state restoration in such a scenario.

**NOTE**

Though we are using two nodes, we will not be able to process more data with this pattern, as the second microservice is used only as a failover. If scalability is needed, we must use the Sequential Convoy pattern and replace each microservice in that pattern with primary and secondary pairs.

**Related patterns**

The following are related to the Two-Node Failover pattern:

*Periodic Snapshot State Persistence and Replay patterns*

These patterns, covered in this chapter, can be used with the Two-Node Failover pattern to restore the state of the microservice when they restart it as a secondary after failure.

*Sequential Convoy pattern*

Used if we need to scale the stream-processing application. This pattern is covered in this chapter.

*Publisher-Subscriber pattern*

To allow both primary and secondary microservices to consume the same events. [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns) details this pattern.

**Summary of Reliability Patterns**

In this section, we outlined some reliability patterns commonly used by cloud native stream-processing applications. [Table 6-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#reliability_patterns-id00262) summarizes when we should and should not use these patterns, and their benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Replay | The system state contains only recent events. To restore state only when there is access to the previously processed events. To process data from persistence stores, filesystems, and log-based message brokers. | We cannot guarantee that previously processed data cannot be accessed again. Dependent systems cannot process duplicate events. Systems cannot take time to re-create their state. The system state needs to contain events that span over a long period. | Allows re-creating state without storing large snapshots. |
| Periodic Snapshot State Persistence | The system state needs to contain events that span over a long period. To restore state only when there is access to the previously processed events. To process data from persistence stores, filesystems, and log-based message brokers. | The system state contains only recent events. We cannot guarantee that previously processed events can be accessed again. Systems cannot take time to re-create their state. | Allows re-creating state faster. Supports larger and long-running system states. Supports dependent applications that cannot process duplicate events. |
| Two-Node Failover | We cannot take time to restore the application after failure. The system state needs to contain events that span over a long period. To restore state only when there is access to the previously processed events. | We cannot guarantee that previously processed events can be accessed again. Systems can take time to re-create their state. | Supports low-latency and highly available stream processing. Supports dependent applications that cannot process duplicate events. |
| Table 6-5. Reliability patterns | | | |

**Technologies**

Stream processing can be designed as part of a microservice, or deployed entirely by using a stream-processing system. When it comes to simple patterns such as Transformation, and Filters and Threshold, we can implement the logic within our microservice. But for more-complex patterns, using a stream-processing technology will provide better outcomes.

In this section, we briefly discuss various stream-processing technologies with the goal of showing how and where we can use them in the design of our cloud native applications.

**Esper**

*Esper* is a complex event-processing library released under the GPL v2 license. It can be used to implement stream-processing logic in Java and .NET-based microservice applications. It supports stream-processing constructs including transformations, filtering, thresholds, windowed aggregations, joins, and temporal event ordering.

Esper can be used to reduce the complexity of an application, as we can offload most of the processing logic to it. We can model events as Java or .NET objects and pass them to Esper for processing, and then subscribe to it to receive outputs. It also supports a query language to configure the stream-processing logic. We recommend using Esper for implementing stream-processing logic within our microservices or serverless functions.

**Siddhi**

*Siddhi* is a Java-based stream-processing library and a microservice released under Apache License v2. As a library, Siddhi (like Esper) can be embedded into microservices to process stream-processing logic. It allows stream-processing logic to be defined via Siddhi Query Language and supports stream-processing constructs including transformations, filtering, thresholds, windowed aggregations, joins, temporal event ordering, and machine learning. We recommend using Siddhi for implementing stream-processing logic within microservices or serverless functions.

We also recommend using Siddhi if you want to run stream processing as a standalone microservice supporting all the reliability patterns, including Periodic Snapshot State Persistence and Two-Node Failover. Users can use Siddhi Query Language to configure the sources from which Siddhi should consume events, its processing logic, and where it should publish its output and deploy that to Kubernetes.

**ksqlDB**

*ksqlDB* is a stream-processing and a database system that is part of Kafka. It works only in environments where Kafka is used as the broker for distributing events. We can define rules in ksqlDB to retrieve events from a Kafka stream, and then process and publish them. It supports stream-processing constructs such as transformations, filtering, thresholds, windowed aggregations, and joins. It also provides a feature to build materialized views from the input events, which can be queried on demand by cloud native applications. Its ability to pull data on demand is useful, as it can be modeled as a relational database. We recommend using ksqlDB when Kafka is used as the message broker for the cloud native application, and when you need to query event logs via materialized views.

**Apache Spark**

*Spark* is a big-data and stream-processing platform released under Apache License v2. It can run on Apache Mesos, Hadoop YARN, and Kubernetes. Though it is strong in batch processing, it can also support stream-processing constructs such as transformations, filtering, thresholds, windowed aggregations, joins, and machine learning.

It uses both queries and a structured programming approach, allowing users to program using Java, Scala, or Python to support both stream and batch processing. It achieves reliable processing by periodically checkpointing data into durable storage. It is an ideal choice when use cases are mainly oriented toward batch processing while having some streaming requirements.

**Apache Flink**

*Flink* is a fully fledged stream-processing platform released under Apache License v2. It can run on platforms like Kubernetes, Knative, and AWS Lambda. It supports stream-processing constructs such as transformations, filtering, thresholds, windowed aggregations, joins, and temporal event ordering, along with graph processing. It also supports exactly once semantics, and supports reliable data processing using watermarks, and snapshots by storing them in storage such as S3, GCS, and HDFS.

Flink supports simple query language for defining stream-processing logic, the Table API for declarative data processing, and data stream and stream-processing APIs in Java for more-granular level configurations. We recommend using this for large-scale stream-processing use cases that have high scalability and availability requirements.

**Amazon Kinesis**

*Kinesis* is a fully managed scalable stream-processing offering from AWS. It supports SQL-based and Flink-based data processing in the cloud and allows users to build their own cloud native applications and run them in Amazon Lambda or EC2. With its SQL mode, it can support transformations, filtering, thresholds, windowed aggregations, and joins.

With Flink, it supports all standard stream-processing constructs. In addition to streaming events, it can also stream video content. We recommend using Kinesis if AWS is the hosting environment for your cloud native application.

**Azure Stream Analytics**

*Azure Stream Analytics* is a fully managed scalable streaming analytics platform offered by Microsoft. It supports defining stream-processing logic by using SQL queries and a graphical user interface. It supports stream-processing constructs such as transformations, filtering, thresholds, windowed aggregations, joins, temporal event ordering, and machine learning.

It also supports hybrid architectures for running stream-processing queries in the cloud and on the edge node. We recommend using Azure Stream Analytics if Azure is your hosting environment.

**Google Dataflow**

*Google Dataflow* is a fully managed scalable stream-processing platform offered by Google. It supports defining stream-processing logic using Apache Beam SDK, SQL queries, and via GUI. It supports stream-processing constructs such as transformations, filtering, thresholds, windowed aggregations, joins, temporal event ordering, and machine learning.

With its Apache Beam SDK, it allows developers to deploy stream-processing logic into on-premises stream-processing systems such as Apache Flink. We recommend using Dataflow if Google Cloud is your hosting environment.

**Summary of Stream-Processing Technologies**

This section outlined commonly used stream-processing systems for cloud native application development. [Table 6-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_technologies) summarizes when we should and should not use these technologies.

| **Stream-processing technology** | **When to use** | **When not to use** |
| --- | --- | --- |
| Esper | To embed into cloud native applications. To support transformations, filtering and thresholds, windowed aggregations, joins, and temporal event ordering. | To run as a standalone application. To run machine learning models. Built-in reliability is required. |
| Siddhi | To embed into cloud native applications. To run as a standalone cloud native application. To support transformations, filtering and thresholds, windowed aggregations, joins, temporal event ordering, and machine learning. | High scalability is needed. |
| ksqlDB | Kafka is used in the infrastructure. To support transformations, filtering and thresholds, windowed aggregations, and joins. To build materialized views from the input events. | Kafka is not used in the infrastructure. Temporal event ordering and machine learning is needed. |
| Apache Spark | Support for both stream and batch processing is needed. To support transformations, filtering and thresholds, windowed aggregations, joins, and machine learning. | A lightweight system is needed for stream processing. Temporal event ordering is needed. To embed into cloud native applications. |
| Apache Flink | To support transformations, filtering and thresholds, windowed aggregations, joins, and temporal event ordering, along with graph processing. For high scalability and availability requirements. | A lightweight system is needed for stream processing. To embed into cloud native applications. |
| Amazon Kinesis | To support Flink in AWS. To support transformations, filtering and thresholds, windowed aggregations, joins and temporal event ordering, along with graph processing. | Other cloud providers are selected. To embed into cloud native applications. |
| Azure Stream Analytics | To support transformations, filtering and thresholds, windowed aggregations, joins, temporal event ordering, and machine learning. To support stream-processing queries to run in the cloud and on the edge node. | Other cloud providers are selected. To embed into cloud native applications. |
| Google Dataflow | To support transformations, filtering and thresholds, windowed aggregations, joins, temporal event ordering, and machine learning. To support portable stream-processing logic that can also run on on-premises stream-processing systems. | Other cloud providers are selected. To embed into cloud native applications. |
| Table 6-6. Stream-processing technologies | | |

**Testing**

In this section, we cover the most important aspects of testing stream-processing cloud native applications. When testing stream-processing applications, we need to follow conventional approaches of writing unit and integration tests. Because stream-processing applications are asynchronous, we recommend you follow all testing suggestions provided for event-driven architecture in [“Testing”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#testing-id00284), along with the suggestions provided here.

One of the key aspects that we need to test in stream-processing applications is their ability to handle state. When we use reliability patterns, we should run chaos testing to test whether the application is continuously producing correct results despite system failures. To assert the application state deterministically, we can use the Watermark pattern to publish watermark events at the end of each test for the application to persist its state and to ensure that it has completed event processing.

When testing time-bounded patterns such as Windowed Aggregations or Temporal Event Ordering patterns, the application could produce different results for each test cycle, due to fluctuations of network and application latency. Instead of asserting the results with a margin of error, we recommend updating the stream-processing applications to process events based on event timestamps generated at the source. With this approach, we can eliminate the network and intermediate application latency and generate reproducible results.

**Security**

How can we enforce security for cloud native stream-processing applications? As discussed in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns), stream-processing applications can enforce security by connecting to message brokers and other systems via secured protocols, and using data and encrypting data in transit and at rest.

If enforcing security at the application level is not possible, we recommend using a bounded context, fronted by an API or a secured message broker to consume the events, and build the whole stream-processing system within that context. We also recommend you apply all the general security best practices discussed in Chapters [2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns) and [5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns).

**Observability and Monitoring**

Observability and monitoring are key for successfully operating stream-processing applications. Because of the asynchronous and stateful nature of the applications, without proper monitoring, we will not be able to detect issues until system failure.

Because stream-processing applications contain state, monitoring their memory consumption is critical. Say a system is running a time-bounded query, such as aggregating values over a five-minute window. If an event spike occurs as the system is storing events in memory, the system could run out of memory, resulting in a failure. Having monitoring and load-shedding mechanisms can help mitigate the risk of abnormal conditions. But if the spike is consistent, we should consider partitioning or remodeling the stream-processing pipeline to cater to higher loads.

Stateful stream-processing systems can utilize the Periodic Snapshot State Persistence pattern to recover their state after failure. Monitoring that these snapshots are properly saved and garbage-collected once outdated is critical. Furthermore, if the application state is large (for example, 500 MB or more), writing the snapshot takes a long time and negatively impacts the available network bandwidth for new events. Therefore, monitoring the size of the snapshots and the time taken to write them, as well as monitoring the network bandwidth, CPU, and memory of the applications when storing the snapshots, will help you properly architect the application to reliably store and restore its state.

The complexity of the stream-processing logic, and network latency, can make events in one stream arrive later than events in other streams. This can cause errors in the final output. Therefore, monitoring what event timestamp is being processed at each streaming application can help identify such discrepancies. We can apply the Buffered Event Ordering, Course Correction, or Watermark pattern to synchronize the applications and mitigate the error.

It is also critical to log and track events by using causation IDs to identify and handle failure situations in production. Refer to [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns) for details on applying them, and for other recommendations on monitoring and observing asynchronous stream-processing applications.

**DevOps**

In this section, we focus on how DevOps applies to stateful applications such as stream-processing applications. The first DevOps step is selecting the appropriate reliability pattern for achieving reliable stream processing. This should be influenced by the system’s availability and scalability requirements.

When a pattern is identified, an appropriate persistence store should be selected to enable the rapid persistence and restoration of state, and store the state durably. We also need to empirically determine the optimal snapshot size and frequency. This should minimize the latency introduced by snapshotting. As part of the DevOps process, we should also monitor and garbage-collect redundant snapshots. Encrypting the snapshots so the sensitive data remains secure is also important.

As discussed in [Chapter 5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_driven_architecture_patterns), we recommend encrypting events when we are dealing with sensitive data, and purging events and snapshots as soon as they are no longer needed for processing. We also recommend using a bounded context when possible, so we can protect all the applications, via API or a topic in the message broker, from external threats.

Because failures in asynchronous applications are difficult to troubleshoot, we also recommend setting up observability and monitoring by using distributed tracing, logging, and monitoring systems. In addition, continuous delivery is crucial for modern DevOps. For smooth deployments, we recommend maintaining backward compatibility of event schema and snapshots at all times. When major changes occur, we recommend running both versions of the applications in parallel until the new system rebuilds its state. Finally, we recommend using multiple deployment environments, such as development and staging/preproduction, to reduce the impact of changes to the application, and to validate the application before moving it to production.

By following these steps, we can safely deploy and maintain cloud native stream-processing applications and systems.

**Summary**

In this chapter, we discussed several stream-processing patterns that can be applied to cloud native applications. We explored options for processing continuous streams of events by using patterns like Transformation, and Filters and Thresholds; using time- and length-based aggregations; joining multiple event streams; detecting interesting incidents based on event occurrence order; and using machine learning to perform real-time predictions.

We covered how to parallelize and scale stream-processing applications, order out-of-order events, synchronize stream-processing operations, and, finally, how to achieve reliable stream processing. We also discussed stream-processing technologies that we can use when applying these patterns, and briefly reviewed how stream-processing applications can be secured, tested, continuously deployed through DevOps, and observed and monitored to guarantee continuous operation. In the next chapter, we will explore the patterns related to API management.